J. A. BAGNELL, B. BOOTS, S. CHOUDHURY

DRAFT: MODERN ADAPTIVE CONTROL AND REINFORCEMENT LEARNING

J.A. BAGNELL, B. BOOTS. AND S. CHOUDHURY

Version 0.1, August 2022

Contents

JAB: My first book was dedicated to my teachers; this one I dedicate to the students from whom I've learned still more.

Prelude

This book is an edited collection of lecture notes from classes given by Drew Bagnell at Carnegie Mellon University in the class *Adaptive Control and Reinforcement Learning* (2010,11,14), from Byron Boots at Georgia Tech (2019) and from Sanjiban Choudhury at Cornell (2022). We thank Chris Atkeson for co-teaching the first instance of this class and shaping how we think about the problems herein. We gratefully acknowledge the many students who took (and put up with!) incomplete notes on the topics covered. We thank Arun Venkatraman who provided the step-by-step derivation of iLQR, and Anqi Li, who provided key editing and improvements to this document.

This book– and the classes it was built from– was designed to provide a set of practical tools to build decision making procedures for machines interacting with the world. Our applications vary from video games and web-search to robot manipulation and self-driving vehicles. The field is vast and so our take is necessarily just one narrow viewpoint. We explicitly make no attempt to be rigorous, but rather focus on intuition and informal mathematical argument to build that intuition, and on techniques we've seen work multiple times on hard decision making problems. We try to outline the techniques and ways of thinking we'd be most likely to pull out in practice. Throughout, we attempt to point to rigorous derivations and the original literature on the topics.

Naturally this work is presented in a somewhat personal context, noting the practical and theoretical differences that arose in implementing real systems. These notes are by no means exhaustive nor is it meant to serve as a summary of the outstanding work in the field; the interested reader will hopefully follow the related reading and bibliography to dive deeper. Instead, it is meant to summarize lessons we've learned, particularly in close collaboration with others in robotics and learning, on problems of making decisions.

These notes were designed to build on essential techniques in probability (conditional probability, conditional independence, Gaussians, integration techniques, Bayesian methods and inference, fil-

tering and time-series models), linear algebra (both computational and basic linear analysis), optimization (gradients, Hessians, metrics, Krylov sub-spaces), and machine learning (generalization, optimization, no-regret/online learning, back-propagation, and kernel methods).

A companion set of lectures notes, covering elements of those techniques particularly in learning and probability, *Statistical Techniques in Robotics* is under development to fill the gap.

1
Markov Decision Problems

1.1 Markov Decision Processes

Overview

We require a formal model of decision making to be able to synthesize and analyze algorithms. In general, making an "optimal" decision requires reasoning about the entire history previous observations, even with perfect knowledge of how an environment works.

A powerful notion that comes to us from the physical sciences is the idea of *state* — a sufficient statistic to predict the future that renders it independent of the past. In classical mechanics, the phase space of positions and momenta forms that state: together with the knowledge of an isolated rigid body (it's inertia) and any torques applied, we can predict the future pose of the object without knowledge of the past.

A *Markov Decision Process* (MDP) is a mathematical framework for modeling decision making under uncertainty that attempts to generalize this notion of a *state* that is sufficient to insulate the entire future from the past. MDPs consist of a set of *states*, a set of *actions*, a deterministic or stochastic *transition model*, and a *reward* or *cost* function, defined below. Note that MDPs do not include observations or an explicit observation model as the environment is assumed to be fully observable at all times: in other words, an agent can observe the state of the world.

The acronym MDP is overloaded to refer to a *Markov Decision Problem* where the goal is to find an optimal *policy* that describes how to act in every state of a given a Markov Decision Process. A Markov Decision Problem includes a notion of what it means for a policy to be optimal, including a *discount factor* that can be used to calculate the present value of future rewards and an *optimization criterion* and a *horizon* (possibly infinite) time that specifies when the problem ends. Strategies for minimizing cost or maximizing reward vary, and

should be time-dependent in finite horizon systems.

The key property – indeed the eponymous property – of an MDP is that it is *Markov*. That is, the probability of observing future states given the past depends (and holding fixed a sequence of actions) only the most recent state and is conditionally independent of the full history. We make that more precise after we introduce notation below to cover key elements of an MDP.

Definitions

1. **State Space:** $x \in \mathbb{X}$ or $s \in \mathbb{S}$. In robotics, examples of state might include the pose of a rover or the configuration of a robot arm. There is typically an initial state, denoted x_0 and possibly a *terminal state* that ends the problem if entered. State is meant to evoke the notion of a full description (like position and velocity in classical mechanics) of the system under consideration that makes the previous trajectory irrelevant to the prediction of the future.

2. **Action:** $a \in \mathbb{A}$ or $u \in \mathbb{U}$. Examples of actions include moving to a discrete neighboring state or torques applied to a joint or wheel. This space is often alternately called the *control* space.

3. **Transition Model:** For stochastic systems, we represent the transition model as the probability of an action a, taken from state x, leading to state x', denoted $x' \sim \mathcal{T}(x, a)$. Here $\mathcal{T}$ can be a probability mass function in case of systems with discrete set of states or a probability density function if the system has a continuous set of states. In deterministic systems, we often explicitly denote the transition model as a deterministic function, i.e., $x' = \mathcal{T}(x, a)$. Note, however, that it is also possible to realize deterministic systems with a stochastic model with the Dirac delta distribution. In an MDP this distribution is well defined, and independent of the past: $p(x'|x, a, \textit{history of all previous } x\textit{'s and } a\textit{'s}) = \mathcal{T}(x, a)$. This is often referred to as the *plant* (particularly in control literature) or the *environment*. We will consider environments that are best modeled by time-varying plans $x' = \mathcal{T}(x, a, t)$ as well in these notes.

4. **Reward or Cost Function:** The reward $r(x, a)$ or cost $c(x, a)$ of taking an action a at a state x. A reward or a cost function can be used interchangeably: we can get the same solution if we define the cost function as the negative of a given reward function and switch the max (for framing as rewards) to min (for framing as costs) during optimization. [1] In some situations, the cost or reward can be a function of only the state s, i.e., $r(x)$ or $c(x)$, or a function of the next state x' after executing action a, $r(x, a, x')$ or $c(x, a, x')$, or some even more complicated combinations like being

[1] Note, however, that sometimes the use of the phrase *cost* is meant to imply that the cost is strictly positive.

also a function of time, *i.e.*, $r(x, a, x', t)$, or can itself be a random variable (i.e., with distribution $p(r \mid x, a, x', t)$). The last form is the most general form that obeys the Markov property and enables efficient computation.

5. **Horizon:** $T \in \mathbb{N}$. The problem is considered over after T steps. This often encodes the number of steps that we care/are able to execute the policy. See Objective Function below. [2]

6. **Discount Factor:** $0 \leq \gamma \leq 1$. This notion determines the current value of future costs or rewards. The intuition is that rewards are more valuable if they happen soon, so if a reward is received n steps in the future, it's only worth γ^n as much as in the present.

7. **Policy:**$\pi \in \Pi : \pi(x, t) = a$. A function that maps states (and an optional time step) into actions. This specifies how to act in any state. [3]

8. **Value Function**: $V^{\pi}(x, t)$. A function used to measure the expected discounted sum of rewards from following a specific policy π from state x. The optimal value function, denoted $V^*(x, t)$, is the value function of the optimal policy π^*, i.e. the policy that yields the highest value for each state x.

9. **Objective Function:** An optimization criteria for a Markov Decision problem.[4] Expected cumulative reward is a common objective function in reinforcement learning:

$$\mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t \, r(x_t, a_t)\right]$$

Other examples include expected infinite discounted reward:

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \, r(x_t, a_t)\right]$$

and immediate reward:

$$\mathbb{E}\left[r(x_0)\right]$$

The goal is to choose a policy that will minimize (if we're using cost functions) or maximize (if we're using reward functions) our objective function. Remember that the policy function just describes the action we take at each time step, so we're effectively finding the best (on average) sequence of actions to complete our task.

To disambiguate some of the notation, from this point on states will be referred to as x, transition models as $\mathcal{T}$, and horizon as T. Because we are pessimistic academics, we will deal in costs c, not rewards.

[2] If $T = 1$, optimal control can be reduced to a greedy search, that is choosing the action with the highest reward. If $0 < T < \infty$, then one must reason T steps ahead to determine the optimal policy starting from the initial state. Often there is no discount factor and the optimal policy may vary wildly as a function of time. The case where $T = \infty$ is typically more likely to converge, as a discount factor γ is used to dampen the effects of oscillation or any time-dependent properties.

[3] In the simplest case, a policy is simply a map from the current state to an action, but policies can be much more general and include information about the transition model or information about the history of previous states ($\pi : \{x_0, ..., x_t\} \times \mathcal{T} \to \mathbb{A}$). We can show that if a decision problem is Markovian, an optimal policy need only be a function of state and time, rather than further history.

[4] Note that optimizing such an objective function does not require the Markov property – that property helps us find policies efficiently.

Example

Consider the simplified game of *Tetris*, where randomly falling pieces must be placed on the game board. Each horizontal line completed is cleared from the board and scores points for the player. The game terminates when the board fills up. The game of Tetris can be modeled as a Markov Decision Process.

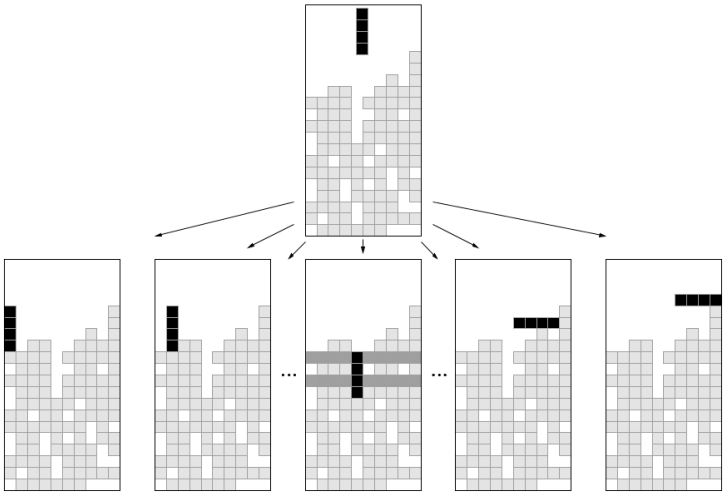


Figure 1.1.1: Example states and transitions for a Tetris scenario with figure from [3].

- **States:** Board configuration (each of k cells can be filled/not filled), current piece (there are 7 pieces total). In this implementation, there are therefore approximately $2^k \times 7$ states. Note: not all configurations are valid, for example, there cannot be a piece floating in the air. This resulting in a smaller number of total valid states.
- **Actions:** A policy can select any of the columns and from up to 4 possible orientations for a total of about 40 actions (some orientation and column combinations are not valid for every piece).
- **Transition Matrix:** A deterministic update of the board plus the selection of a random piece for the next time-step.
- **Cost Function:** There are several options to choose from, including: reward = $+1$ for each line removed, 0 otherwise; # of free rows at the top; $+1$ for not losing that round; etc.

Deterministic and Non-Deterministic MDP Algorithms

For Deterministic MDPs the transition model is deterministic or, equivalently, we know with certainty what the next state x' will be given the current state x and the action a. Solving deterministic

MDPs is often traditionally posed as a search problem. There are many approaches to solving deterministic MDPs using search, many of which are much more efficient than generic MDP approaches. Here are three flavors of approach that one might try:

1. The Greedy Approach: choose the action at the current time that minimizes the immediate cost.
2. Naive Exhaustive Search: explore every possible action for every possible state and choose the series of actions that minimizes the total cost.
3. Pruning: Search possible actions, but remember only the cheapest series of actions, ignoring the previously discovered paths with higher cost.

A naive exhaustive search is often computationally ineffective as its complexity is $O(\exp(T))$.

An exhaustive search can produce the optimal policy at the expensive of high computational (and sample complexity) cost. While the greedy approach is often cheap to compute, it may sometimes produce policies that are not remotely good. The pruning approach balances the computational cost and the quality of the resulting policies. Often it can produce a reasonably good policy in a much shorter time compared to the exhaustive search algorithms. However, if we care to find *the optimal policy*, then we need to consider all policies.

Non-deterministic problems, where the next system state is not known with certainty, naturally suggest considering the expectation of future rewards for any given action. One strategy, called *Value Iteration*,[5] discussed in the later section, calculates the expected sum of discounted rewards for each state under the optimal policy (the *value* of that state, denoted V^*, also known as the optimal value function) without explicitly computing the optimal policy. An optimal policy can then just act by greedily selecting the action with the highest value. Some alternatives will be covered later in the course, such as *Policy Iteration* and *Q-Learning*. Policy iteration evaluates a given policy then improves upon the policy and repeats the process. We latter consider methods which do not require an apriori known transition model, and instead attempt to use samples of state-action pairs to compute an optimal action from any state.

[5] The Value Iteration algorithm is also applicable to deterministic MDPs. In fact, we will see how to use Value Iteration to solve deterministic MDPs in the next section.

1.2 Solving MDPs

Scenario

Let's consider the case where a robot is traversing a known maze-like environment from a start location to a goal location. The environ-

ment is discretized into a 2D grid. Actions are movements in the cardinal directions. The cost is $+1$ (a unit of "suffering") for being in every state except for the goal state where the cost is 0. The goal is a terminal "absorbing" state, so once our robot achieves the goal state it cannot leave – the robot has achieved nirvana and the suffering is over. Our task is to choose a sequence of actions that take the robot from the start state to the goal state while minimizing the expected total cost. In other words, we want to minimize

$$\mathbb{E}\left[\sum_{t=0}^{T-1} c(x_t, a_t)\right]$$

We'll first look at a deterministic problem where the robot will move to the adjacent cell in the direction of the action if the cell is free: there may be obstacles or walls in the grid, in which case the robot is unable to transition into those states. In this simple deterministic problem, with the cost for each state except for the goal being 1, the optimal value at each state is simply the minimum number of states traversed to get from that state to the goal. The optimal policy returned at each cell is then the direction the robot should travel to minimize the number of steps needed to reach the goal.

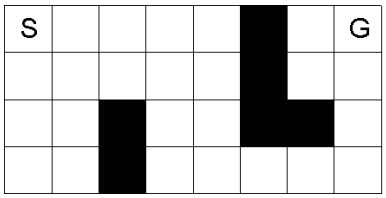


Figure 1.2.1: Discrete World, Start (S), Goal (G). Obstacles are denoted by the black squares

Dynamic Programming Formulation for Solving Deterministic MDPs

Time T − 1:

We can write this in a straightforward recursive formulation of this problem, we start at the last timestep, $t = T - 1$. Here, the optimal policy is just choosing the action with the minimal cost and the value function at each state is the minimum cost of all actions from a given location.

$$\pi^*(x, T-1) = \operatorname*{argmin}_a c(x, a)$$
$$V^*(x, T-1) = \min_a c(x, a)$$

Time T − 2:

Now the values at the last timestep are the same everywhere except at the goal. Next, consider the next-to-last step $t = T - 2$. Suppose that we are at state x and we take action a, the total cost would

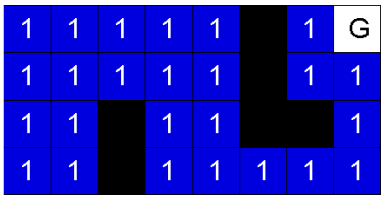


Figure 1.2.2: Optimal Value Function for each state at time $T-1$

be the value of the next state $x' = \mathcal{T}(x,a)$ at the last timestep $T-1$ plus the immediate cost of taking action a in our current state x. Therefore, we should simply choose an action a that minimizes the sum these two terms. The optimal value of each state is then the minimum of the cost of the action a at current state x and plus the optimal value of the next state x' at the last timestep $T-1$.

$$\pi^*(x, T-2) = \operatorname*{argmin}_a \left[c(x,a) + V^*(x', T-1)\right]$$
$$V^*(x, T-2) = \min_a \left[c(x,a) + V^*(x', T-1)\right]$$

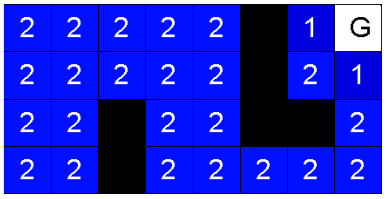


Figure 1.2.3: Optimal Value Function for each state at time $T-2$

Time T − 3 and below

We can define a general recursion to calculate the optimal value and optimal policy functions. For any given time $t \leq T-2$, we have:

$$\pi^*(x,t) = \operatorname*{argmin}_a \left[c(x,a) + V^*(\mathcal{T}(x,a), t+1)\right]$$
$$V^*(x,t) = \min_a \left[c(x,a) + V^*(\mathcal{T}(x,a), t+1)\right]$$

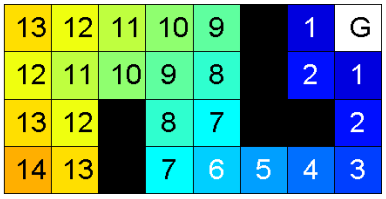


Figure 1.2.4: Final value function after T steps of Value Iteration

We can also write recursive algorithms that produce the optimal value and the optimal policy for any state, at any time t, considering a T-length time horizon. Algorithm 1 below describes the recursive method that computes the best value function (cost-to-go) for a given

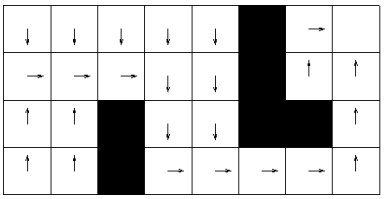


Figure 1.2.5: Action at each location using the final policy.

state x starting at time t and stopping at time $T-1$.

Algorithm 1: Recursive algorithm for computing the optimal value function

Algorithm `OptimalValue`(x, t, T)
  if $t = T-1$ **then**
    return $\min_a c(x,a)$
  end
  else
    return $\min_a c(x,a) +$ `OptimalValue`$(\mathcal{T}(x,a), t+1, T)$
  end

How do we compute the best policy? One important concept we can observe from the algorithms above is that if we use the optimal value function we never need to explicitly compute the optimal policy. Policy and value are *not* the same, but if the optimal value function is given, the optimal policy can be easily recovered, as shown below:

$$\pi^*(x,t) = \operatorname*{argmin}_a \left[c(x,a) + V^*(\mathcal{T}(x,a), t+1)\right].$$

But what if we want to get the optimal policy *while* computing the optimal value? Let's first define an auxiliary algorithm that returns the value function with time horizon T for a given policy π, starting at state x. This is called *policy evaluation* and is described in Algorithm 2.

Algorithm 2: Policy evaluation: a recursive algorithm that computes the value function for a given policy

Algorithm `Value`(x, π, t, T)
  if $t = T-1$ **then**
    return $c(x, \pi(x,t))$
  end
  else
    return $c(x, \pi(x,t)) +$ `Value`$(\mathcal{T}(x, \pi(x,t)), \pi, t+1, T)$
  end

The above can, of course, be implemented as an in-place dynamic program by starting from the last time-step as in Algorithm 3 as

describe in the equations above for the robot problem.

We can also extract via a *dynamic program* (backwards induction) that proceeds from the last time step Algorithm 2 to compute the optimal policy $\pi^*(x,t)$ for all states and time steps:

Algorithm 3: Algorithm for computing the optimal policy

Algorithm `OptimalPolicy(`x, T`)`
- **for** $t = T-1,\ldots,0$ **do**
  - **for** $x \in \mathbb{X}$ **do**
    - **if** $t = T-1$ **then**
      - $\pi^*(x,t) = \underset{a}{\operatorname{argmin}}\, c(x,a)$
    - **end**
    - **else**
      - $\pi^*(x,t) =$
      - $\underset{a}{\operatorname{argmin}}\, c(x,a) +$ `Value(`$\mathcal{T}(x,a), \pi^*, t+1, T$`)`
    - **end**
  - **end**
- **end**

Note that the complexity of computing the optimal policy via the dynamic program above is $O(|\mathbb{X}||\mathbb{A}|T^2)$. However, because we are repeatedly calculating many of these function calls, we can *memoize* previously computed value functions (i.e. from future time steps) resulting in an algorithm with complexity $O(|\mathbb{X}||\mathbb{A}|T)$. Below, we'll explictly use backwards induction to create *Value Iteration*, the "industry standard" efficient means to compute the optimal value function rather than rely on ad-hoc memoization.

It is worth noting that the value function is a function of time. You might see why by considering, for instance, a hockey game, in which a team's actions may vary widely depending on the time remaining. If a team is losing and there are seconds left, they may choose to pull their goalie off the ice and have an extra scoring player. At the start of the game, even if losing, pulling the goalie is generally a very unwise decision.

Backwards Induction Formulation for Solving General MDPs

Consider now MDPs that are not deterministic– that is, problems with uncertainty in the transition model. Here we will consider opti-

mizing the expectation over the optimal value function:

$$\begin{aligned}
\pi^*(x,t) &= \operatorname*{argmin}_a \left[c(x,a) + \mathbb{E}\left[V^*(x',t+1)\right]\right] \\
&= \operatorname*{argmin}_a \left[c(x,a) + \sum_{x'} p(x'|a,x)\, V^*(x',t+1)\right], \\
V^*(x,t) &= \min_a \left[c(x,a) + \mathbb{E}\left[V^*(x',t+1)\right]\right] \\
&= \min_a \left[c(x,a) + \sum_{x'} p(x'|a,x)\, V^*(x',t+1)\right].
\end{aligned}$$

Applying backwards induction (dynamic programming) instead of a recursive formulation, we get what is known as *Value Iteration*:

Algorithm 4: Dynamic Programming Value Iteration for computing the optimal value function.

Algorithm `OptimalValue(`x, T`)`
- **for** $t = T-1, \ldots, 0$ **do**
  - **for** $x \in \mathbb{X}$ **do**
    - **if** $t = T-1$ **then**
      - $V(x,t) = \min_a c(x,a)$
    - **end**
    - **else**
      - $V(x,t) = \min_a c(x,a) + \sum_{x' \in \mathbb{X}} p(x'|x,a) V(x,t+1)$
    - **end**
  - **end**
- **end**

This approach now has complexity $O(|\mathbb{X}|^2|\mathbb{A}|T)$. However, since we often don't have to sum over all $x \in \mathbb{X}$ as the probability of transitioning to those states may be 0, this typically reduces to $O(k|\mathbb{X}||\mathbb{A}|T)$, where k is the average number of neighbouring states. In a deterministic problem, of course $k = 1$. If our environment is continuous, the sums above become integrals as we are integrating over the state space.

Infinite Horizon Problems

Recall that when we have a finite horizon, both the optimal value function and the optimal policy are functions of time. However, as T approaches infinity, we expect that the optimal value function and the optimal policy no longer have such dependence on time. Consider, for example, the maze problem above: we would expect the value function to stabilize as the horizon T gets large. Similarly, it would seem surprising to alter our policy at different time steps

when there is no time limit (imagine a game that lasts forever). In some cases, the value function (optimal, or for a given policy) will **not** converge in the infinite horizon case. Typically, failure of convergence for the infinite horizon problem is caused by divergence (for example, when the goal is unreachable), but oscillation of the value function can also prevent the value function from converging. A simple example of the oscillation problem is shown below:

Exercise: Construct examples that lead to value function divergence. Relate to the classical convergence criteria for series in sequences in college-level calculus.

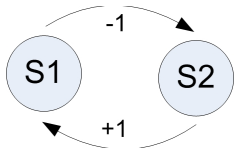


Figure 1.2.6: Value Function Oscillation

If the value function does converge, we are assured a stationary feedback policy that is optimal. [6]

[6] Exercise: Why? Make the argument.

Rewards and Discount Factors

Thus far, we have only talked about cost functions in our examples. Instead, imagine using a **reward** function, where the robot gets zero points for each move, unless it moves into the goal, whereupon it receives 100 points. You can see that there is very little urgency for the robot to move towards the goal, as it can spend as many steps as it wants wandering the state space before reaching the goal while still receiving the same 100 points.

In order to avoid situations like this, we can apply the *discount factor* mentioned above. Since discount factors value obtaining rewards sooner rather than later, they incent the robot to move to the goal as quickly as possible.

More morbidly, discount factors can alternatively be thought of as a way of contending with the possibility of death. [7] Under this interpretation, at each time step, the robot lives with probability γ, and dies with probability $(1-\gamma)$ (goes to an absorbing state that has 0 reward or value). The optimal value function then becomes:

[7] Understanding a discount factor as imposing an effective horizon of $O(\frac{1}{1-\gamma})$ and understanding it as being the result of a transition to a terminal state are often powerful ways to reason intuitively about algorithms and analysis in optimal decision making.

$$\begin{aligned} V^*(x,t) &= \min_a \left[c(x,a) + \sum_{x'} \left[\gamma \left[p(x'|a,x)\, V^*(x', t+1) \right] + (1-\gamma) \times 0 \right] \right] \\ &= \min_a \left[c(x,a) + \gamma \sum_{x'} p(x'|a,x)\, V^*(x', t+1) \right] \end{aligned}$$

The fixed point version of the above equation (i.e., what we would expect to hold as the finite horizon value function to converge as $T \to \infty$) is called the **Bellman equation**.

$$V^*(x) = \min_a \left[c(x,a) + \gamma \sum_{x'} p(x'|a,x)\, V^*(x') \right]$$

We will explore this equation in more detail below.

Convergence and Optimal Solutions

If $\gamma < 1$, we can guarantee that the sum of rewards achieved by the agent is finite with probability 1 (assuming the reward is as well for each state and time) and that the optimal value function will converge. For many special cases, the value function will also converge for $\gamma = 1$, but this is not generally true for the reasons we discussed above.

It is important to bear in mind that once the value converges, it– and the optimal policy– becomes invariant with relation to the time.[8]

[8] Exercise: Convince yourself this must be true.

$$V^*(x,t) \xrightarrow{t\to\infty} V^*(x) = \min_a \left[c(x,a) + \gamma \sum_{x'} p(x'|x,a)\, V^*(x') \right]$$

And the same happens for the optimal policy:

$$\pi^*(x,t) \xrightarrow{t\to\infty} \pi^*(x) = \operatorname*{argmin}_a \left[c(x,a) + \gamma \sum_{x'} p(x'|x,a)\, V^*(x') \right]$$

There are two iterative approaches for finding this convergence value.

Approach 1 In this approach, we define a small threshold ε (this could be interpreted as a as a confidence level) and we will run the algorithm for a time horizon that is sufficiently large so that the error in that value will be of magnitude $O(\varepsilon)$. Choosing T such that $\gamma^T = O(\varepsilon)$, i.e. $T = O(\log(\frac{1}{\varepsilon}))$, ensures that our error is $O(\varepsilon)$. We then simply run Algorithm 4 for T time-steps, use execute the resulting (time-varying!) policy. [9]

[9] It's unclear what to do in this approach when the policy executes T or more steps. Cycling the policy again could be a reasonable procedure but is ad-hoc. Of course, theoretically it doesn't matter because times larger than T, by construction, are exponentially damped in their significance.

Algorithm 5: Dynamic Program for creating an optimal value function on the infinite horizon by finite horizon approximation

Algorithm `OptimalValue`(x, T)
  for $t = T-1, \ldots, 0$ **do**
    for $x \in \mathbb{X}$ **do**
      if $t = T-1$ **then**
        $V(x,t) = \min_a c(x,a)$
      end
      else
        $V(x,t) = \min_a c(x,a) + \gamma \sum_{x' \in \mathbb{X}} p(x'|x,a) V(x,t+1)$
      end
    end
  end

Approach 2 Alternately one can use an iterative, in-place method, based on the Bellman equation, where the result obtained in one step is plugged back into the equation until it converges.

Algorithm 6: Iterative approximation algorithm

for $x \in \mathbb{X}$ **do**
  $V(x) = \min_a c(x, a)$
while *does not converge* **do**
  for $x \in \mathbb{X}$ **do**
    $V^{new}(x) = \min_a c(x, a) + \gamma \sum_{x' \in \mathbb{X}} p(x'|x, a)\, V^{old}(x')$
  $V^{old}(x) \leftarrow V^{new}(x), \forall x$
return $V^{new}(x), \forall x$

Both algorithms will return the optimal value function for all states as the number of iterations tends to infinity. As mentioned earlier, once the value function is known, it is possible to obtain the policy. Thus, these algorithms also allow us to obtain the optimal policy for every state.

Approach 1 can be demonstrated to have theoretically stronger performance bounds if we execute the time-varying policy that results rather than keeping only the value and policy computed at $t = 0$, perhaps intuitively as it is actually the optimal solution for the finite horizon problem.[10] Approach 2 is not the optimal solution for *any* specific problem but rather is an approximate iterative method. Nevertheless, Approach 1 can be costly: it requires a considerable amount of extra memory, since it keeps track of *all* future values for each given time step. Approach 2 initializes the value function V and iteratively finds better approximations of that value by plugging its current value into the solution equation. Compared with the first approach, this approach has a slower convergence rate as a function of the number of iterations in the worst case, but requires a smaller amount of memory. One can also consider simple variants (covered in [Puterman, 1994]) that maintain a single value functions and update data *in place*.[11]

[10]

[11] Similar to a *Gauss-Seidel* method for solving linear systems.

1.3 *Related Reading*

[1] Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. Probabilistic Robotics. Cambridge, MA: MIT, 2005. Ch 14, pp 499-502 for most relevant material.

[2] Andrew Moore's slides: http://www.autonlab.org/tutorials/mdp.html

[3] Boumaza, A. How to design good Tetris players, Tech Report, University of Lorraine, LORIA, 2014.

[4] Puterman, M. Markov Decision Processes: Discrete Stochastic Dynamic Programming, 2005.

2
LQR: The Analytic MDP

2.1 The Linear Quadratic Regulator

In the previous chapter we defined MDPs and investigated how to compute the value function and optimal policy at any state with Value Iteration. While the examples thus far have involved discrete state and action spaces, important applications of the basic algorithms and theory of MDPs include problems where both states and actions are continuous. Perhaps the simplest such problem is the *Linear Quadratic Regulator* (LQR) problem.

LQR techniques are one of the most effective and widely used methods in robotics and control systems design. The basic problem is to identify a mapping from states to controls that minimizes the quadratic cost of a linear (possibly time varying) system. A quadratic cost has the form,

$$c(x, u) = x^\top Q\, x + u^\top R\, u, \tag{2.1.1}$$

where $x \in \mathbb{R}^n$ is the state of the system, and $u \in \mathbb{R}^k$ is the control.[1] In the cost function, Q should be symmetric positive semi-definite ($Q = Q^\top, Q \succeq 0$).[2] It does not have to be strictly positive definite in general.[3] For example, in the cart-pole balancing problem, we only need the pendulum to stay upright and we do not care much about where the cart is. However, to avoid infinite control effort, R should be strictly positive definite ($R = R^\top, R \succ 0$).

[1] Precisely the same as *action a* in the previous section. Here we choose to use u to denote actions in order to be consistent with the broad literature on control.

[2] Why? Note what would fail if Q did not have these properties? Is symmetry a requirement?

[3] For instance because sometimes we do not require *every* component of the state to reach 0 and don't care about these components

Exercise

There is disagreement in the literature as to what *positive definite* applied to a matrix Q means: does it imply symmetry, or just that $x^\top Q\, x > 0$ for all non-zero x? Let's see the root of this confusion: Why can we consider $Q = Q^\top$ without any loss of generality in the LQR problem? Specifically, make the opposite assumption, and then consider a symmetric Q that would lead to precisely the cost

function. In a sense then, we can simply assume positive definiteness implies symmetry as this is simpler to countenance and will lead to equivalent results.

Continuous Control of a Discrete-Time System

An example of a continuous time-invariant system with quadratic cost is the problem of balancing a simple inverted pendulum. The pendulum is illustrated in Figure 2.1.1. The simple pendulum consists of a bob, modeled as a point mass, and attached to a mass-less rigid rod. Let the mass of the bob be m, the length of the rod be l, and gravity be g. The angle between the pendulum and the y-axis θ is controlled by the torque τ exerted at the origin. The dynamics of this system is given by

$$\begin{aligned} ml^2\ddot{\theta} &= mgl\sin\theta + \tau \\ \Rightarrow \ddot{\theta} &= \frac{g}{l}\sin\theta + \frac{1}{ml^2}\tau \\ &\approx \frac{g}{l}\theta + \frac{1}{ml^2}\tau \end{aligned} \tag{2.1.2}$$

To find the control policy of the system, we first linearize it about

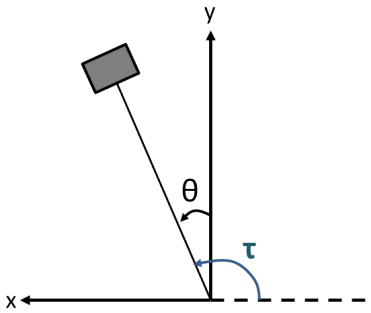


Figure 2.1.1: An inverted pendulum.

the up-right configuration. Let $\alpha = g/l$, and assume $ml^2 = 1$. The state space equations become

$$\begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}_{t+1} = \begin{bmatrix} 1 + \frac{1}{2}\Delta t^2 \cdot \alpha & \Delta t \\ c \cdot \Delta t & 1 \end{bmatrix} \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}_t + \begin{bmatrix} \frac{1}{2}\Delta t^2 \\ \Delta t \end{bmatrix} \tau \tag{2.1.3}$$

The optimal control policy can be found by formulating an MDP. For the linearized simple pendulum,

- state: $x_t = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}_t$,
- action: $u_t = \tau$,
- cost: $c(x, u) = x^\top Q x + u^\top R u$,

- dynamics: $x_{t+1} = Ax_t + Bu_t$,
  where $A = \begin{bmatrix} 1 + \frac{1}{2}\Delta t^2 \cdot c & \Delta t \\ c \cdot \Delta t & 1 \end{bmatrix}$ and $B = \begin{bmatrix} \frac{1}{2}\Delta t^2 \\ \Delta t \end{bmatrix}$.

We already know how to solve this problem: Value Iteration! Let's look at this more closely.

2.2 *Value Iteration for Linear Quadratic MDPs*

Let the value function of the MDP for a finite-horizon problem with horizon T be be $J^\pi(x_t, t)$, i.e.

$$J^\pi(x_t, t) = \sum_{t'=t}^{T-1} c(x_{t'}, \pi(x_{t'}, t')). \tag{2.2.1}$$

Recall the **Bellman Equation** for the finite horizon problem:

$$\begin{aligned} J^*(x, t) &= \min_{u_t} \left[c(x_t, u_t) + J^*(x_{t+1}, t+1)\right] \\ &= \min_{u_t} \left[({x_t}^\top Q x_t + {u_t}^\top R u_t) + J^*(x_{t+1}, t+1)\right] \end{aligned} \tag{2.2.2}$$

and

$$J^*(x, T-1) = \min_{u_{T-1}} \left[c(x_{T-1}, u_{T-1})\right] \tag{2.2.3}$$

Let's consider the recursive formulation for solving this problem.

Time T − 1:

At the last time step $t = T - 1$, the solution to Equation 2.2.2 is $u_{T-1} = 0$. This is due to the fact that we are not concerned with the next step since we already reach the time limit. Hence, *any* action will increase the cost: minimizing (2.2.3) is essentially minimizing $u_{T-1}^\top R u_{T-1}$. By definition, R is a positive definite matrix, and therefore setting $u_{T-1} = 0$ can result in minimum cost at $t = T - 1$.

Now let's calculate the optimal value function $J^*(x_{T-1}, T-1)$. Since ${u_{T-1}}^\top R u_{T-1} = 0$, by (2.2.3),

$$J^*(x_{T-1}, T-1) = {x_{T-1}}^\top Q x_{T-1} \doteq {x_{T-1}}^\top V_{T-1} x_{T-1}, \tag{2.2.4}$$

where V_{T-1} is the *value matrix*.[4]

[4] A common alternate notion is to use P_t instead of V_t to denote the value matrix.

In summary, at the last time step, we have a zero control and a value that is quadratic in the state.

Time T − 2:

The optimal value function at $t = T - 2$ is,

$$J^*(x_{T-2}, T-2) = \min_{u_{T-2}} c(x_{T-2}, u_{T-2}) + J^*(x_{T-1}, T-1) \tag{2.2.5}$$

$$= \min_{u_{T-2}} \left({x_{T-2}}^\top Q x_{T-2} + {u_{T-2}}^\top R u_{T-2} + {x_{T-1}}^\top V_{T-1} x_{T-1}\right). \tag{2.2.6}$$

For the sake of notational simplicity, let $x = x_{T-2}$ and $u = u_{T-2}$. From the dynamics of the system, $x_{T-1} = Ax + Bu$.

$$J^*(x, T-2) = \min_u \left\{ x^\top Q x + u^\top R u + (Ax + Bu)^\top V_{T-1}(Ax + Bu) \right\} \tag{2.2.7}$$

Taking the partial derivative of the function to be minimized with respect to u and setting it to 0 yields

$$\begin{aligned} 2Ru + 2B^\top V_{T-1} Ax + 2B^\top V_{T-1} Bu &= 0 \\ (R + B^\top V_{T-1} B)u &= -B^\top V_{T-1} Ax \\ u &= -(R + B^\top V_{T-1} B)^{-1} B^\top V_{T-1} Ax \end{aligned} \tag{2.2.8}$$

The solution to u always exists because the inverse of $R + B^\top V_{T-1} B$ exists since R is positive definite and $B^\top V_{T-1} B$ is at least positive semi-definite. Let $K_{T-2} = -(R + B^\top V_{T-1} B)^{-1} B^\top V_{T-1} A$,

$$u_{T-2} = K_{T-2} x_{T-2}. \tag{2.2.9}$$

The control u_{T-2} is a linear function of state x_{T-2} with control matrix K_{T-2}. The optimal value function at $t = T-2$ can be found as

$$\begin{aligned} J^*(x_{T-2}, T-2) &= {x_{T-2}}^\top Q x_{T-2} + {x_{T-2}}^\top K_{T-2}^\top R K_{T-2} x_{T-2} \\ &\quad + {x_{T-2}}^\top (A + BK_{T-2})^\top V_{T-1}(A + BK_{T-2}) x_{T-2} \\ &= {x_{T-2}}^\top (Q + K_{T-2}^\top R K_{T-2} + (A + BK_{T-2})^\top V_{T-1}(A + BK_{T-2})) x_{T-2} \\ &\doteq {x_{T-2}}^\top V_{T-2} x_{T-2}. \end{aligned} \tag{2.2.10}$$

Observe that in this time step, the value is *also* quadratic in state. Therefore, we can derive similar results of linear control and quadratic value for every time step prior to $t = T-2$:

$$\begin{aligned} K_t &= -(R + B^\top V_{t+1} B)^{-1} B^\top V_{t+1} A \\ V_t &= \underbrace{Q}_{\text{current cost}} + \underbrace{K_t^\top R K_t}_{\text{cost of action at t}} + \underbrace{(A + BK_t)^\top V_{t+1}(A + BK_t)}_{\text{cost to go}} \end{aligned} \tag{2.2.11}$$

and the optimal value function is,

$$J^*(x_t, t) = {x_t}^\top V_t x_t. \tag{2.2.12}$$

Algorithm 7 summarizes value iteration for LQRs:

Algorithm 7: LQR value Iteration

Algorithm `OptimalValue`(A, B, Q, R, t, T)
  if $t = T - 1$ **then**
    return Q
  end
  else
    $V_{t+1} =$ `OptimalValue`$(A, B, Q, R, t+1, T)$
    $K_t = -(R + B^\top V_{t+1} B)^{-1} B^\top V_{t+1} A$
    return $V_t = Q + K_t^\top R K_t + (A + BK_t)^\top V_{t+1}(A + BK_t)$
  end

The complexity of the above algorithm is a function of the horizon T, the dimensionality of the state space n, and the dimensionality of the action space k: $O(T(n^3 + k^3))$.

Convergence of Value Iteration

What about the infinite horizon version of the LQR problem? That is, we are considering

$$J^\pi(x_t) = \sum_{t=0}^{\infty} c_t(x_t, \pi(x_t, t)). \tag{2.2.13}$$

Recall that in the finite horizon LQR problem, K_t and V_t are computed backward in time as,

$$\begin{aligned} K_t &= -(R + B^\top V_{t+1} B)^{-1} B^\top V_{t+1} A \\ V_t &= Q + K_t^\top R K_t + (A + BK_t)^\top V_{t+1}(A + BK_t). \end{aligned} \tag{2.2.14}$$

One natural idea is to keep applying (2.2.14) until K_t and V_t converge to a fixed point. The associated question is thus, do K_t and V_t always converge? And further, if they do not always converge, when do they actually converge? The answer is, K_t and V_t converge if the system is so called *stabilizable*,[5] and they converge to the solution to the Discrete Algebraic Ricatti Equation (DARE): [6]

$$\begin{aligned} V &= Q + K^\top R K + (A + BK)^\top V (A + BK) \\ K &= -(R + B^\top V B)^{-1} B^\top V A \end{aligned} \tag{2.2.15}$$

Moreover, the K and V that solve the DARE indeed yield the optimal policy for the infinite horizon LQR problem. We can view V as a combination of the cost of current state and control, along with the future cost. If the system is not stabilizable, for example, a system of two motors controlling two inverted pendulums with one of the motors broken, then K_t and V_t no longer converge. However, the

[5] Brian D. O. Anderson and John B. Moore. *Optimal Control: Linear Quadratic Methods*. Prentice-Hall, Inc., 1990

[6] The conditions for LQR to converge are effectively identical to that of any other value iteration problem. It's enough here that we can asymptotically drive all the state variables to 0.

value iteration will still return the policy that can get the system to work as well as possible by using the good motor to attempt to stabilize the system. On the infinite horizon, it may, of course lead to a diverging value estimate– in essence, the issues that happen in finite state spaces with non-converging value functions can happen in solving the Ricatti equation. [7]

[7] There are linear-algebraic methods to solve the Ricatti equations as well as simply the natural Value-Iteration backup procedure; these can be more computationally efficient, but are rarely required

2.3 *Extensions of LQR*

In the following sections, we continue to expand the domain of applicability of the general strategy for solving LQR problems developed above. The basic techniques that we will augment LQR with include

1. Allowing the system to be time varying
2. Allowing general affine systems (via homogenous coordinates or direct derivation)
3. Moving from controls to "deviations" in control
4. Iteratively re-linearizing

We visit each of these incrementally, as it's useful to see each addition, and end up with a general algorithm for a wide class of control problems.

Tracking Trajectories with LQR

The method described in Algorithm 7 will not work for a pendulum "swing up" problem, since the system dynamics at $\theta = 0^\circ$ (unstable) and $\theta = 180^\circ$ (stable) are *qualitatively* different. Linearization will fail as the linearized model (2.1.3) is a good approximation of the non-linear dynamics only at a small region around $\theta = 0^\circ$.

Given a trajectory, possibly recorded from an expert demonstration, (x_t, u_t) from $\theta = 180^\circ$ to $\theta = 0^\circ$ (see Fig 2.3.1), one might imagine that it could simply be replayed to balance the inverted pendulum. However, this doesn't work in practice due to modeling error– moreover, the same sequence of controls is unlikely to produce exactly the same behavior when played twice on a real system due to minor variations in the system. However, a reference trajectory can still be useful. One way to use an expert trajectory in presence of uncertainty, is to use LQR *tracking*, which we describe below. Before describing how tracking works, we first introduce several minor variations on the LQR approach, including LQR for *Linear Time Varying* dynamical systems, *Affine Quadratic Regulation*, and LQR with stochastic dynamics.

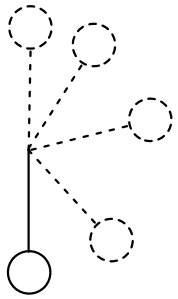


Figure 2.3.1: Solving inverted pendulum swing up using LQR tracking.

LQR for Linear Time-Varying Dynamical Systems

Thus far, we have assumed that we were modeling a linear, time-invariant system. As we will see, we might be interested in systems that are linear, but time varying

$$x_{t+1} = A_t x_t + B_t u_t \tag{2.3.1}$$
$$c(x_t, u_t) = {x_t}^\top Q_t x_t + u_t^\top R_t u_t \tag{2.3.2}$$

In this case, the LQR equations are simply updated to

$$K_t = -(B_t^\top V_{t+1} B_t + R_t)^{-1} B_t^\top V_{t+1} A_t \tag{2.3.3}$$
$$V_t = Q_t + K_t^\top R_t K_t + (A_t + B_t K_t)^\top V_{t+1}(A_t + B_t K_t) \tag{2.3.4}$$

Affine Quadratic Regulation

Let's now consider a generic *affine* system with time varying dynamics A_t and B_t and a state *offset* x_t^{off}:

$$x_{t+1} = A_t x_t + B_t u_t + x_t^{\text{off}}. \tag{2.3.5}$$

Affine problems can be converted to linear problems by using *homogeneous coordinates*[8]:

$$\tilde{x} = \begin{bmatrix} x \\ 1 \end{bmatrix} \tag{2.3.6}$$
$$\tilde{x}_{t+1} = \begin{bmatrix} A_t & x_t^{\text{off}} \\ 0 & 1 \end{bmatrix} \tilde{x}_t + \begin{bmatrix} B_t \\ 0 \end{bmatrix} u_t \doteq \tilde{A}_t \tilde{x}_t + \tilde{B}_t u_t \tag{2.3.7}$$

This is just a new LQR problem with modified state and dynamics and a new cost defined as $c(\tilde{x}_t, u_t) = \tilde{x}_t^\top \tilde{Q}_t \tilde{x}_t + u_t^\top R_t u_t$, where the choice of $\tilde{Q}$ is problem dependent. We will later see how we can design $\tilde{Q}$ for the tracking problem. The Affine Quadratic Regulation problem can then be solved in exactly the same way as the LQR problem.[9]

[8] https://en.wikipedia.org/wiki/Homogeneous_coordinates

[9] Essentially the same trick can be applied to enable us to have linear cost functions terms in the controls as well, but we defer this to the general formulation derived at the end.

Tracking

There are two natural formulations for a tracking cost function:

$$c_t(x_t, u_t) = (x_t - x_t^*)^\top Q(x_t - x_t^*) + (u_t - u_t^*)^\top R(u_t - u_t^*) \tag{2.3.8}$$
$$c_t(x_t, u_t) = (x_t - x_t^*)^\top Q(x_t - x_t^*) + u_t^\top R u_t \tag{2.3.9}$$

where x_t^* and u_t^* are the nominal trajectory and nominal control input obtained from the expert (not necessarily optimal ones!). Q penalizes the deviation from the nominal trajectory and R penalizes either the deviation from the nominal controls or is just a penalty on the control (*e.g.* lots of actuation is bad).

Expanding the term corresponding to state error in the cost function:

$$\begin{aligned}(x_t - x_t^*)^\top Q(x_t - x_t^*) &= x_t^\top Q x_t + \underbrace{x_t^{*\top} Q x_t^*}_{\text{constant at time } t} - \underbrace{2x_t^{*\top} Q}_{\text{constant at time } t} x_t \\ &= x_t^\top Q x_t + d_t - 2q_t^\top x_t,\end{aligned}$$

where $d_t \doteq x_t^{*\top} Q x_t^*$ and $q_t \doteq Q x_t^*$. Next, we choose a $\tilde{Q}_t$ defined as:

$$\tilde{Q}_t = \begin{bmatrix} Q & -q_t \\ -q_t^\top & d_t \end{bmatrix},$$

such that the state error term of the cost function can be formulated as $\tilde{x}_t^\top \tilde{Q}_t \tilde{x}_t$, where $\tilde{x}_t$ is the homogeneous coordinates in (2.3.6). Note that d_t is a constant, which only shifts the cost function in an uninteresting way.

For cost functions with the control error term of the form $(u_t - u_t^*)^\top R(u_t - u_t^*)$, let $\tilde{u}_t = (u_t - u_t^*)$. Then the corresponding term of the cost function can be modified as $\tilde{u}_t^\top R \tilde{u}_t$. In order to use $\tilde{u}_t$ instead of u_t in the cost function defined as in Eq. 2.3.8, the dynamics needs to be modified as follows:

$$\tilde{x}_{t+1} = \begin{bmatrix} A_t & x_t^{\text{off}} + B_t u_t^* \\ 0 & 1 \end{bmatrix} \tilde{x}_t + \begin{bmatrix} B_t \\ 0 \end{bmatrix} \tilde{u}_t. \tag{2.3.10}$$

The modified cost function is:

$$c_t(\tilde{x}_t, \tilde{u}_t) = \tilde{x}_t^\top \tilde{Q}_t \tilde{x}_t + \tilde{u}_t^\top R \tilde{u}_t. \tag{2.3.11}$$

Solving the LQR for the system using the above cost function

$$\tilde{u}_t = -\tilde{K}_t \tilde{x}_t.$$

Subsequently u_t is obtained as $u_t = \tilde{u}_t + u_t^*$.

2.4 *Iterative LQR (iLQR)*

So far, we have seen how to use LQR to solve problems with linear (or affine) dynamics and quadratic costs. However, real world systems will only rarely be close to linear. [10]

Differential Dynamic Programming (DDP) [11] is a general approach to using quadratic approximations of the value function to solve a broader class of control problems than merely linear-Gaussian. *Iterative LQR* (iLQR) is a simplified variant of DDP, an approach that repeatedly solves LQR (actually affine!) problems to solve for a locally optimal change to a trajectory and a controller around that. The idea of iLQR is very closely related to Newton's method (where we first approximate the objective function to a quadratic function, minimize it, and iterate until convergence). In iLQR, we first approximate the dynamics with an affine model and approximate the cost function with a quadratic function. Crudely speaking, we then solve the LQR problem for the resulting approximate problem, and iterate the process until convergence.

[10] There is a well-known saying among control theorists
> Classifying systems as linear and nonlinear is like classifying the Universe as bananas and non-bananas.

[11] D. H. Jacobson and D. Q. Mayne. *Differential Dynamic Programming*. Elsevier, 1970.

The algorithm

The general iLQR strategy is as follows:

1. Propose some initial (feasible) trajectory $\{x_t, u_t\}_{t=0}^{T-1}$
2. Linearize the dynamics, f about trajectory:

$$\left.\frac{\partial f}{\partial x}\right|_{x_t} = A_t, \quad \left.\frac{\partial f}{\partial u}\right|_{u_t} = B_t$$

   Linearization can be obtained by three methods:

   (a) Analytical: either manually or via *auto-diff*, compute the correct derivatives.

   (b) Numerical: use finite differencing.

   (c) Statistical: Collect samples by deviations around the trajectory and fit linear model.

3. Compute second order Taylor series expansion the cost function $c(x, u)$ around x_t and u_t and get a quadratic approximation $c_t(\tilde{x}_t, \tilde{u}_t) = \tilde{x}_t^\top \tilde{Q}_t \tilde{x}_t + \tilde{u}_t^\top \tilde{R}_t \tilde{u}_t$ where the $\tilde{x}_t, \tilde{u}_t$ variables represent *changes* in the proposed trajectory in homogenous coordinates. [12]
4. Given $\{A_t, B_t, \tilde{Q}_t, \tilde{R}_t\}_{t=0}^{T-1}$, solve an affine quadratic control problem and obtain the proposed feedback matrices (on the homogeneous represenation of x).

[12] We haven't derived using homogeneous coordinates in control; it's essentially equivalent to simply completing the square and finding a "nominal" control. Instead of pursuing yet another step-wise generalization, in the appendix to these notes presents the general derivation.

5. Forward simulate the full nonlinear model $f(x, u)$ using the computed controls $\{u_t\}_{t=0}^{T-1}$ that arise from feedback matrices applied to the sequence of states $\{x_t\}_{t=0}^{T-1}$ that arise from that forward simulation.
6. Using the newly obtained $\{x_t, u_t\}_{t=0}^{T-1}$ repeat steps from 2.

Issues with iLQR

- Q and R can be indefinite when the actual cost function is not convex. Hacks that are typical in the literature include:
  - Projection: $Q = U \underbrace{\Sigma}_{\text{set negative Eigenvalues to 0}} U^\top$. Formally, this can be shown as finding the closest (in L_2 sense cost matrix that actually is PSD.
  - Regularize: Increase the diagonal values until Q becomes positive definite: $Q = Q + \lambda I$
- Trust regions: Sometimes the approximation of the cost function is poor and in such cases its a good idea to restrict the step size (deviation from the trajectory of the previous iteration) while executing the control. This can be accomplished in the following ways:
  - interpolate between the control at current iteration and the previous iterations
  - Modify cost to penalize derivation from the trajectory of the previous iteration:

    $\tilde{c} = c + \alpha \cdot (\text{penalty for deviation from the previous trajectory in controls or states})$

    These last known as *control* and *state* damping are extremely common in real-world implementations.
- Some notes:
  LQR recieved significant practical criticism in the 1970s as it was difficult to prevent the resulting synthesized controllers from exciting dynamics that were under-modeled. Without care, LQR (particularly using filtered estimates of the true state, rather than "oracle" access to the true state) will often generate, "stiff", high frequency controls that are not robust. Some common modifications to damp high frequency control from being generated include:
  - Penalize *changes* in control from previous control. This is to ensure that the control is smooth. Higher order of smoothness can be obtained by passing the control signal through a filter,

modeled in the system dynamics, and then using the output of that as "effective" control input for the system.

– More generally, we can implement a *filter* on the execute control dynamics by storing previous controls in the state vector and penalizing any linear operation on these.

It's often useful to model latency by a simple "loading" controls into states by including that delay in the dynamics:

$$\begin{bmatrix} x_t \\ u_{t-1} \end{bmatrix} = \begin{bmatrix} A & B \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_{t-1} \\ u_{t-2} \end{bmatrix} + \begin{bmatrix} 0 \\ u_{t-1} \end{bmatrix}. \tag{2.4.1}$$

This method for modeling delay is crude but effective. More sophisticated approaches include providing an *immutable* region of controls are often used in *receeding* horizon control.

2.5 *Differential Dynamic Programming (DDP)*

The original, fancier version of approximate value iteration for locally linear quadratic systems is called differential dynamic programming (DDP). iLQR and DDP are very similar, the difference being that iLQR assumes a simpler linear model for the system dynamics, while DDP uses a full quadratic model and then truncates any terms that are higher than second order in the value function expansion. The result is that DDP provides a correct-to-second-order expansion of the value function. iLQR is slightly simpler to implement than DDP and often provides similar or better results empirically for less computation.

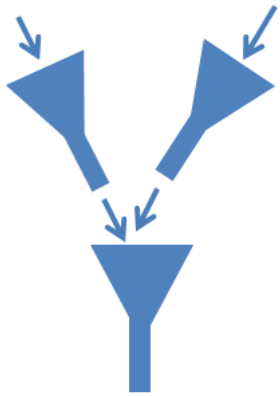


Figure 2.5.1: Funnels can be a metaphor for controllers, and you can think of composing funnels that cover different parts of the space of states.

DDP (or iLQR) builds a second order approximation of the value function, giving a quadratic bowl at every timestep. This ends up acting like a series of funnels [2]. When you are in the area covered

by a funnel, you are pulled toward the optimum. You can think of composing funnels such that one funnel dumps you out into another funnel. If you cover the entire space with funnels, then you can imagine that each one is a controller that is good in a certain section of the space (See Figure 2.5.1). With iLQR, we built a quadratic value function about a particular trajectory, but you can imagine starting somewhere else. If you can get from that starting point into the region covered by your value function, then you already know what to do from there. Chris Atkeson wrote a classic paper on this subject, in which he looks at covering the state space with DDP policies [3]. Imagine an inverted pendulum: there will be some controller that is good for the near-vertical case. One can then have other controllers covering other parts of the space, and each controller gets closer to the set of states it knows how to handle, funneling states towards the goal.

LQR with Stochastic Dynamics

The treatment of the Linear Quadratic Control problem up until now has assumed that the dynamics of the system are deterministic: the next state of the system can be determined precisely from the previous state and the control input.

$$x_{t+1} = Ax_t + Bu_t \tag{2.5.1}$$

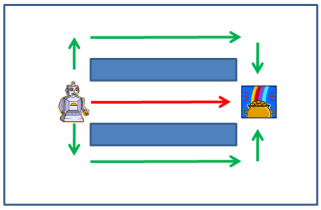


Figure 2.5.2: Robot in grid world showing optimal policy for deterministic (red) vs. stochastic (green) motion.

It is not at all clear, however, that the policy built for the deterministic case is the policy that you would follow if you knew there was noise. For example, imagine a robot in a grid world (See Figure 2.5.2). The robot is positioned on the opposite side of two obstacles from the goal (pot of gold). Hitting an obstacle is catastrophic for the robot, but there is just enough space for the robot to drive between the two obstacles to reach the goal. If the robot's motion is deterministic, then the best policy is to drive between the obstacles. But if the robot's motion is stochastic, and with a 10% probability, the robot moves in a random direction instead of the commanded direction, then the best policy is to avoid walking the tightrope between the dangerous obstacles, and to instead go around the obstacles.

We can extend LQR to handle a simple case of stochastic dynamics and derive the optimal policy for this case. We will assume that at each time step, a zero mean Gaussian perturbation affects the state [13].

$$x_{t+1} = Ax_t + Bu_t + \varepsilon_t \tag{2.5.2}$$

where $\varepsilon_t \sim \mathcal{N}(0, \Sigma)$. x_{t+1} can also be written as

$$x_{t+1} \sim \mathcal{N}(Ax_t + Bu_t, \Sigma) \tag{2.5.3}$$

[13] Note that the noise that we are adding is motion model noise. We are not considering a non-trivial observation model here.

Recall that for the deterministic case, the optimal policy at time t, π_t^*, is given by finding the action that minimizes the sum of action cost and cost-to-go from the resulting state

$$\pi_t^* = \underset{u_t}{\operatorname{argmin}}\, c(x_t, u_t) + J^*(x_{t+1}, t+1) \tag{2.5.4}$$

The problem is that in the stochastic case, the next state x_{t+1} can not be predicted exactly. As with value iteration, the solution is to replace the optimal cost-to-go J^* by the expected value of J^* given the previous state and selected action. The expression for π_t^* thus becomes

$$\pi_t^* = \underset{u_t}{\operatorname{argmin}}\, c(x_t, u_t) + \mathbb{E}\left[J^*(x_{t+1}, t+1)\right] \tag{2.5.5}$$

The expectation term in this expression is the integral

$$\mathbb{E}\left[J^*(x_{t+1}, t+1)\right] = \int_{\mathbb{X}} x_{t+1}^\top V_{t+1} x_{t+1} \mathcal{N}(x_{t+1}; Ax_t + Bu_t, \Sigma)\, \mathrm{d}x_{t+1} \tag{2.5.6}$$

This integral belongs to a class of integrals called Gaussian Integrals and has a simple closed form solution.

$$\int (x-b)^\top P(x-b)\mathcal{N}(x; \mu, \Sigma)\, \mathrm{d}x = (\mu - b)^\top P(\mu - b) + \operatorname{Tr}\left[P\Sigma\right] \tag{2.5.7}$$

substituting we get

$$\mathbb{E}\left[J^*(x_{t+1}, t+1)\right] = (Ax_t + Bu_t)^\top V_{t+1}(Ax_t + Bu_t) + \operatorname{Tr}\left[V_{t+1}\Sigma\right] \tag{2.5.8}$$

or, since $J^*(x_t, t) = x_t^\top V_t x_t$,

$$\mathbb{E}\left[J^*(x_{t+1}, t+1)\right] = J^*(Ax_t + Bu_t, t+1) + \operatorname{Tr}\left[V_{t+1}\Sigma\right] \tag{2.5.9}$$

Thus using the expectation of the optimal cost-to-go in the stochastic case gives almost the same expression as using the value of cost-to-go in the deterministic case. The only difference is the trace term which is a constant when Σ is fixed or depends only on t. Since all the values under the argmin are shifted by the same constant value, the policy will remain unchanged by the presence of noise, even though the value function has changed. The new trace term added to the cost-to-go can be considered the cost incurred due to uncertainty.

It should be emphasized that this analysis only holds when Σ is independent of the control u. In many real settings, this does not hold. For example, on a robot, the larger the motion the larger the induced uncertainty in position is.

2.6 Related Reading

[1] Y Tassa, T Erez, E Todorov. "Synthesis and stabilization of complex behaviors through online trajectory optimization." IEEE/RSJ International Conference on Intelligent Robotcs and Systems, 2012

[2] R. R. Burridge, A. A. Rizzi, and D. E. Koditschek, "Sequential composition of dynamically dexterous robot behaviors." International Journal of Robotics Research, 18(6):534–555, June 1999.

[3] C. G. Atkeson, "Using Local Trajectory Optimizers to Speed Up Global Optimization", Proceedings of Neural Information Processing Systems, December 1993.

2.7 Appendix: Derivation of the General ILQR Backup steps

The following provides a detailed derivation of the iLQR approach. At each iteration of the algorithm, we execute a proposed current policy to get a trajectory. That we compute the dynamic program below to provide an update to that policy. This is iterated until convergence.

Given the true dynamics F, we can find the Taylor expansion around a proposed trajectory $(x_t\star, u_t\star)$:

$$\begin{aligned} x_{t+1} &= A(x_t - x_t\star) + B(u_t - u_t\star) + F(x_t\star, u_t\star) \\ \Rightarrow x_{t+1} - x_{t+1}\star &= A(x_t - x_t\star) + B(u_t - u_t\star) \\ z_{t+1} &= Az_t + Bv_t \end{aligned}$$

where we define $z_t = x_t - x_t\star$ as the change to the state trajectory and $v_t = u_t - u_t\star$ as the change to the control trajectory.

Similarly, given the true cost function C, the second order taylor expansion is:

$$\begin{aligned} c_t(x_t, u_t) &= \frac{1}{2}[(x_t - x_t\star)^T, (u_t - u_t\star)^T] H \begin{bmatrix} (x_t - x_t\star) \\ (u_t - u_t\star) \end{bmatrix} + g^T \begin{bmatrix} (x_t - x_t\star) \\ (u_t - u_t\star) \end{bmatrix} + C(x_t\star, u_t\star) \\ &= \frac{1}{2}[(x_t - x_t\star)^T, (u_t - u_t\star)^T] \begin{bmatrix} Q & P \\ P^T & R \end{bmatrix} \begin{bmatrix} (x_t - x_t\star) \\ (u_t - u_t\star) \end{bmatrix} + [g_x^T, g_u^T] \begin{bmatrix} (x_t - x_t\star) \\ (u_t - u_t\star) \end{bmatrix} + C(x_t\star, u_t\star) \\ &= \frac{1}{2}(x_t - x_t\star)^T Q(x_t - x_t\star) + (x_t - x_t\star)^T P(u_t - u_t\star) + \frac{1}{2}(u_t - u_t\star)^T R(u_t - u_t\star) + g_x^T(x_t - x_t\star) + g_u^T(u_t - u_t\star) + c \end{aligned}$$

and thus that we can right down a cost function in the *changes* to state/action as:

$$\Rightarrow c(z_t, v_t) = \frac{1}{2} z_t^T Q z_t + z_t^T P v_t + \frac{1}{2} v_t^T R v_t + g_x^T z_t + g_u^T v_t + c$$

Dynamic Programming (Value-Iteration) Backup

Assume we have now a control policy of the form of a "feedforward" update term k_t and feedback term K_T that is a linear controller response to "errors" in z_T:

$$v_T = K_T z_T + k_T, \tag{2.7.1}$$

Inductively, we assume the next-state value function (i.e. of the future timestep) can be written in the form,

$$J_{T+1} = \frac{1}{2} z_{T+1} V_{T+1} z_{T+1} + G_{T+1} z_{T+1} + W_{T+1}. \tag{2.7.2}$$

Since

$$\begin{aligned} z_{T+1} &= A z_T + B v_T && (2.7.3)\\ &= A z_T + B(K_T z_T + k_T) && (2.7.4)\\ &= (A + BK_T) z_T + B k_T, && (2.7.5) \end{aligned}$$

we can write, J_{T+1} as:

$$\begin{aligned} J_{T+1} &= \frac{1}{2}((A+BK_T)z_T + Bk_T)^T V_{T+1}((A+BK_T)z_T + Bk_T) + G_{T+1}((A+BK_T)z_T + Bk_T) + W_{T+1} && (2.7.6)\\ &= \frac{1}{2} z_T^T (A+BK_T)^T V_{T+1}(A+BK_T) z_T + \frac{1}{2} k_T^T B^T V_{T+1} B k_T + k_T^T B^T V_{T+1}(A+BK_T) z_T && (2.7.7)\\ &\quad + G_{T+1}(A+BK_T) z_T + G_{T+1} B k_T + W_{T+1} && (2.7.8)\\ &= \frac{1}{2} z_T^T (A+BK_T)^T V_{T+1}(A+BK_T) z_T + \left(k_T^T B^T V_{T+1}(A+BK_T) + G_{T+1}(A+BK_T) \right) z_T && (2.7.9)\\ &\quad + G_{T+1} B k_T + \frac{1}{2} k_T^T B^T V_{T+1} B k_T + W_{T+1} && (2.7.10) \end{aligned}$$

Additionally, we can write the cost $c_T(z_T, v_T)$ as:

$$\begin{aligned} c_T &= \frac{1}{2} z_T^T Q z_T + z_T^T P v_T + \frac{1}{2} v_T^T R v_T + g_x^T z_T + g_u^T v_T + c + J_{T+1} && (2.7.11)\\ &= \frac{1}{2} z_T^T Q z_T + z_T^T P(K_T z_T + k_T) + \frac{1}{2}(K_T z_T + k_T)^T R(K_T z_T + k_T) + g_x^T z_T + g_u^T(K_T z_T + k_T) + c && (2.7.12)\\ &= \frac{1}{2} z_T^T Q z_T + z_T^T P K_T z_T + k_T^T P^T z_T + \frac{1}{2} z_T^T K_T^T R K_T z_T + \frac{1}{2} k_T^T R k_T + k_T^T R K_T z_T + g_x^T z_T && (2.7.13)\\ &\quad + g_u^T K_T z_T + g_u^T k_T + c \\ &= \frac{1}{2} z_T^T \left(Q + 2PK_T + K_T^T R K_T \right) z_T + \left(k_T^T P^T + k_T^T R K_T + g_x^T + g_u^T K_T \right) z_T + \frac{1}{2} k_T^T R k_T + g_u^T k_T + c && (2.7.14) \end{aligned}$$

Then, we can write $J_T = c_T(z_T, v_T) + J_{T+1} = \frac{1}{2} z_T^T V_T z_T + G_T z_T + W_T$ by combining like terms from above, where

$$\begin{aligned} V_T &= Q + 2PK_T + K_T^T R K_T + (A+BK_T)^T V_{T+1}(A+BK_T) && (2.7.15)\\ G_T &= k_T^T P^T + k_T^T R K_T + g_x^T + g_u^T K_T + k_T^T B^T V_{T+1}(A+BK_T) + G_{T+1}(A+BK_T) && (2.7.16)\\ W_T &= \frac{1}{2} k_T^T R k_T + g_u^T k_T + c + G_{T+1} B k_T + \frac{1}{2} k_T^T B^T V_{T+1} B k_T + W_{T+1} && (2.7.17) \end{aligned}$$

We find the control policy by minimizing J_T with respect to v_T.

$$\begin{aligned} v_T &= \min_{v_T} c_T + J_{T+1} && (2.7.18)\\ &= z_T^T P v_T + \frac{1}{2} v_T^T R v_T + g_u^T v_T + \frac{1}{2}(A z_T + B v_T)^T V_{T+1}(A z_T + B v_T) + G_{T+1}(A z_T + B v_T) && (2.7.19)\\ &= \left(z_T^T P + z_T^T A^T V_{T+1} B \right) v_T + (G_{T+1} B + g_u^T) v_T + \frac{1}{2} v_T^T \left(R + B^T V_{T+1} B \right) v_T && (2.7.20)\\ & && (2.7.21) \end{aligned}$$

Taking the derivative with respect to v_T and setting equal to 0, we get,

$$0 = \left(P^T + B^T V_{T+1} A\right) z_T + (B^T G_{T+1}^T + g_u) + \left(R + B^T V_{T+1} B\right) v_T \tag{2.7.22}$$

$$v_T = -\left(R + B^T V_{T+1} B\right)^{-1} \left(P^T + B^T V_{T+1} A\right) z_T - \left(R + B^T V_{T+1} B\right)^{-1} (B^T G_{T+1}^T + g_u) \tag{2.7.23}$$

$$= K_T z_T + k_T \tag{2.7.24}$$

where $K_T = -\left(R + B^T V_{T+1} B\right)^{-1} \left(P^T + B^T V_{T+1} A\right)$ and $k_T = -\left(R + B^T V_{T+1} B\right)^{-1} (B^T G_{T+1}^T + g_u)$.

Plugging this resulting policy back in to the expression for V_T, G_T and W_T completes the dynamic programming by providing us a quadratic form for the value function. (Note that W_T and c are actually irrelevant as they are constants in the optimization)

3
Receding Horizon Control and Practical Trajectory Optimization

4
Practical Optimization: Constraints and Games

4.1 Introduction to Games and Constraints

This lecture focuses on a class of techniques for managing constraints in optimization problems that arrive either in machine learning or control. The below set of techniques are ones we've found quite helpful in practice.

There are a few general classes of approach to managing constraints and the details of which is "best" depends on requirements on speed and on quality of the required constraint satisfaction and final cost. There are strong inter-relations between them, but certain ones are usually preferred. The lecture below assumes an equality constraint, but each result can be derived for inequality constraints.

4.2 Reparameterization

Arguably the "simplest" technique, when viable, is reparameterization—*i.e.* reformulate the variables so the constraint **must** be satisfied. We might call that the *physics way*: if temperature is empirically lower bounded, re-expresses in $1/T$, or if velocity is upper-bounded, re-express dynamics to enforce relativistic constraints. This is also the critical technique of the beautiful theory of Generalized Linear Models, where, *e.g.*, we express binomial probabilities as unconstrained logits pushed through a logistic function. It's also fundamental to the "shooting" methods of trajectory optimization and control like iLQR, where dynamic constraints are explicitly formulated by forwarding simulating controls (that may exist only to satisfy the reparameterization).

Reparameterization naturally allows us to express uncertainty over, e.g. controls, because we can use the Laplace approximation in the reparameterization to get meaningful uncertainty estimates. [1] For instance, in a control application, if we reparameterize a positive-only velocity components as $\text{speed} = e^{\text{log\_speed}}$, we can produce posterior confidence intervals on speed that satisfy the constraints as well by passing through the reparameterization. Reparameterization equally allows feedback gains in techniques like iLQR to continue to be useful.

Slow convergence can result as we get close (or initialize unfortunately close), as the effect of reparameterization (and a cost on that variable) becomes like a barrier function and leads to very ill-conditioned problem and

[1] B. D. Ziebart, J. Andrew Bagnell, and A. K. Dey. Modeling interaction via the principle of maximum causal entropy. In *Proceedings of the 27th International Conference on Machine Learning*, 2010

effectively cost "plateaus". We have seen this behavior as a general issue with interior point methods, although that observation is a frequent point of contention. Also, for optimal control problems, reparamertization typically buries more non-linearity in the state transition dynamics and this may be ignored by some methods (e.g. iLQR, in contrast with classical Differential Dynamic Programming).

4.3 Lagrange (Primal-Dual) Methods

We'll begin by considering the problem of minimizing $f(x)$ subject to the constraint $g(x) = 0$.

Lagrange methods convert constraints into a saddle point problem– *i.e.* a game where the "dual" (multiplier player) attempts to take advantage of any violation of constraints. We must find solutions where the dual player can't win, and if we do, we guarantee satisfying the constraints. A surprising range of practical methods for constrained optimization look like "apply some simple optimization strategy to the dual problem". [2]

[2]

Gradient/ExpGrad on multipliers

Defining the dual:

$$D(\lambda) = \min_x f(x) - \lambda^T g(x),$$

it's straightforward to take

$$\max_\lambda D(\lambda)$$

and simply compute sub-gradients (assuming, e.g. $f(x)$ is convex, otherwise we're estimating a still more general notion of derivative and an application of Danskin's theorem) of D. Note this is "easy" as $\nabla_\lambda \min_x L(x, \lambda) = \nabla_\lambda L(x^*, \lambda)$ where $x^* = \text{argmin}\, f(x) - \lambda^T g(x)$. (Where this is being evaluated at some specific λ!)

We can then simply run our favorite derivative based optimizer (heavy-ball momentum, sub-gradient, Exponentiated Gradient Descent, etc.) on this dual. We note the problem may very well *not* be strongly concave in λ: that is, we may have essentially no curvature in λ– necessitating slow step size decreases, controlled/stable optimization in general, and potentially even uphill steps. [3]

[3] We note in passing the the subgradient method is equivalent to two related, natural algorithms: a) Iteratively linearizing the objective function and providing a shrinking regularization to 0. b) Iteratively linearizing the objective function and regularizing to the previous location.

These are special cases of two different, stable, optimization strategies (a) FTRL (Follow the Regularized Leader) and (b) Mirror Descent (Proximal Descent/Trust Region) that are popular in game solving and online learning, and under linearization happen to collapse to the same classical gradient descent.

To see (b), note:

$$\text{argmin}_x \nabla f(x_{\text{last}})^T x + \frac{\alpha}{2}||x - x_{\text{last}}||^2$$

is solved by,

$$x = x_{\text{last}} - \frac{1}{\alpha}\nabla f(x_{\text{last}}),$$

which is gradient descent with learning rate $\frac{1}{\alpha}$.

Regularized Lagrangian Optimization / Penalty Methods

A natural way to frame the optimization problem in light of the potential instability in optimizing the dual is to consider the regularization (see side note) that is implicit in gradient descent. That is, we can consider the *Regularized Lagrangian* (ReLa):

$$max_\lambda D(\lambda) + \frac{\alpha}{2}\lambda^2$$

This form ensures compact level sets for the optimization and lets us actually solve the problem without solutions in λ running away to infinity.

I'll assume below that with regularization we can safely assume strong duality for the problem. (That's very much unnecessarily strong, as we can converge to a dual solution anyway, but it saves time/thought.)

Given strong duality, we can swap the min and max and solve the dual:

$$\min_x \max_\lambda f(x) - \lambda^T g(x) - \frac{\alpha}{2}\lambda^2$$

But note that the inner maximization is now *closed-form*. That is, if compute ∇_λ, we get back:

$$\lambda = -\frac{1}{\alpha} g(x)$$

Substituting in, we can now eliminate λ and solve the unconstrained problem:

$$\min_x f(x) + \frac{1}{2\alpha} g(x)^2$$

That's particularly interesting, as what's classically known as a *penalty method* arises as simply the optimal solution of a regularized version of the Lagrangian; penalty methods are Lagrange methods in disguise. Shrinking the Lagrange regularization is simply scheduling the penalty, and we can see that this is a sound method, and can even compute the sub-optimality of the constraints from it. Finally, it provides an estimate of the dual variables directly via the relation $\lambda = -\frac{1}{\alpha} g(x)$.

The only reason not to take regularization to 0 is that we have poor conditioning and extremely large gradients that which implies optimization methods are likely suffer in the way barrier methods do— but it's super fast and easy to implement.

Note further that we can regularize the Lagrange multipliers in many ways– for instance, by (unnormalized) entropy regularization. Each leads to a dual penalty method (in this case an exponential penalty method). Performance of these depends a great deal on the underlying optimizer as well as our prior beliefs on the multipliers λ. If the Lagrange multipliers are expected to have small L_2 norm, we would expect classical squared-norm regularization to be good. Entropy regularization, by contrast, we might expect will work well when there are many constraints, but a sparse set of multipliers are sufficient to satisfy the constraints. Adapting the literature on such dual penalty methods is likely valuable to taking best advantage of these techniques.

L_∞ ReLa / "Exact" Penalty

An interesting exercise is to consider the Regularized Lagrangian using the $\alpha \max_i |\lambda_i|$ as the regularizer. A quick calculation reveals that this leads to a *penalty method* that is solving:

$$\min_x f(x) + \frac{1}{\alpha}|g(x)|$$

This form is related to that given by the Support Vector Machine primal, where the constraint we attempt to enforce correct labelings. The classic hinge-loss here arises from an inequality exact penalty method on achieving the correct label, although this is not usually how the method is described. In practice, non-trivial regularization of the dual variables in an SVM is assumed to prevent over-fitting.

Note however, that this form, while excellent for sub-gradient based optimizers (AdaGrad, etc.) popular in machine learning, fails to be strongly convex (or smooth) and thus is poorly optimized by Newton-style methods. The advantage of this method is that a *finite value* of $\frac{1}{\alpha}$ is sufficient to achieve the constraint precisely in contrast with squared norm penalty approach.

Augmented Lagrangian Optimization / Mirror Descent

Using Mirror Descent, where we iteratively regularize around the last solution for the dual variable rather than around 0, provides a powerful generalization of penalty methods. We begin with an estimate of the Lagrange multipliers, λ_0. As with ReLa above, we consider a regularized dual of the Lagrangian:

$$\min_x \max_\lambda f(x) - \lambda^T g(x) - \frac{\alpha}{2}(\lambda - \lambda_i)^2$$

Note again that the inner optimization (with x fixed) is closed form (in terms of the next λ to choose):

$$\lambda = \lambda_i - \frac{1}{\alpha} g(x)$$

and by substitution we can eliminate λ and solve the regularized dual as a simpler optimization in x:

$$\min_x f(x) - \lambda_i^T g(x) + \frac{1}{2\alpha} g(x)^2$$

In contrast with the penalty method, we can simply iterate. We update λ with

$$\lambda_{i+1} = \lambda_i - \frac{1}{\alpha} g(x),$$

and re-solve the Augmented Lagrangian (AuLa) [4] problem above.

[4]

It's interesting to ask why we might bother with this form, when the penalty method is itself sufficient. A general theme in optimization is that it can be more efficient to phrase a problem as a saddle-point-finding exercise *rather* than as a difficult, pure optimization. AuLa takes the penalty method approach and moves it back towards a game, managing to inherit both the simplicity of each as well as better conditioning than the single optimization solved in a penalty method. The general strategy of replacing a hard optimization problem with a game-derived sequence of such is a powerful area of research and the more detailed variant of that, the strategy of regularizing the dual parameters [5] to ensure closed form solution is one that seems not fully exploited in the literature.

[5]

Inequality Variant of Mirror Descent on Lagrangian

Perhaps the classic way to handle inequalities in the Lagrange methods above is to add a slack variable with a bound constraint $z \geq 0$. However, we can actually derive variants directly using the technique above as well.

We'll begin by considering the problem of minimizing $f(x)$ subject to the constraint $g(x) \geq 0$. Following the logic above and applying duality, we can form a regularized or augmented Lagrangian as:

$$\min_x \max_{\lambda \geq 0} f(x) - \lambda^T g(x) - \frac{\alpha}{2}(\lambda - \lambda_i)^2$$

Solving for λ (note this happening component-wise in λ and I'm glossing over that) gives us,

$$\lambda = \max(0, \lambda_i - \frac{g(x)}{\alpha})$$

Eliminating λ, we have, again component-wise, but I'm too lazy to index into λ and thus I'm treating it like it's a scalar/single constraint: If $(\lambda_i - \frac{g(x)}{\alpha}) >$

0, we are minimizing

$$f(x) - \lambda_i^T g(x) + \frac{g(x)^2}{2\alpha}$$

and otherwise,

$$f(x) - \frac{\alpha \lambda_i^2}{2}$$

This result is continuous and differentiable, but it is not twice so, so **YMMV**. The primal problem can also have achieve a minima with a negative objective value, which is somewhat annoying. Again we iterate the algorithm by re-estimating

$$\lambda_{i+1} = \max(0, \lambda_i - \frac{g(x)}{\alpha})$$

and solving again for x.

4.4 Projected Gradient

A remarkably simple method, which is often very useful in machine learning, is to take a gradient step and then project (in the sense of finding the nearest location in the constraint set in Euclidean norm) onto the constraint set. If constraints are simple and convex, this method will *accelerate* convergence relative to the unconstrained problem, at least for convex losses. [6] This is because we guarantee each projection step takes us closer to the optima in the set.

This method fundamentally relies upon the projection step being simple: for instance onto a unit ball or with bound constraints. Both of these are trivially implemented as normalization and thresholding, respectively. These style of constraints are quite common on ML applications like support vector machines (SVMs) [7]. Unfortunately, the simplicity of the *Projected Gradient* approach is compromised when using methods (for instance, Gauss-Newton or AdaGrad) that rely on a *non-trivial metric*: the projection must take place using the same metric (*i.e.* the same notion of closeness) to ensure good behavior. One can convince one self of counter-examples to proper convergence using Euclidean projection combined with Newton steps.

[6] M. Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *Proceedings of the 20th International Conference on Machine Learning (ICML)*, 2003

[7] N. Ratliff, J. A. Bagnell, and M. Zinkevich. (Online) subgradient methods for structured prediction. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2007

4.5 Related Reading

[1] Bertsekas, D. Nonlinear Programming: 3rd Edition. Athena Scientific, 2016.

[2] Toussaint, M. *A tutorial on Newton methods for constrained trajectory optimization and relations to SLAM, Gaussian Process smoothing, optimal control, and probabilistic inference.* In Geometric and Numerical Foundations of Movements, Springer, 2017.

5
Policy Iteration

We saw in previous notes that value iteration can be used to find an optimal policy efficiently. Interestingly, in later rounds of value iteration, the best action at each state rarely changes. Put in other words, the policy implicitly defined by the value function appears to converge more rapidly than the value function itself. This insight suggests approaches that attempt to update an explicit estimate of the optimal policy rather than only an explicit estimate of optimal value function (with the policy implicit).

Policy Evaluation

In order to update the policy, we need some way to measure its performance. Fortunately, we have a way to do this: we can simply compute the value function for a fixed policy. We can use the value function $V^\pi(x,t)$ to denote the expected cost-to-go of a policy π in state x at time t. The process of finding V^π is called *policy evaluation*. We can use a policy evaluation algorithm to tell us how good one policy is to compare to others or suggest modifications.

Recall from the first chapter that we can compute the value function for a given policy π through the following algorithm:

Algorithm 8: Dynamic Program for creating an optimal value function on the infinite horizon by finite horizon approximation

Algorithm `EvaluatePolicy(`x, π,T`)`
  for $t = T-1,\ldots,0$ **do**
    for $x \in \mathbb{X}$ **do**
      if $t = T-1$ **then**
        $V(x,t) = c(x,\pi(x,t))$
      end
      else
        $V(x,t) = c(x,\pi(x,t)) + \gamma \sum_{x' \in \mathbb{X}} p(x'|x,\pi(x,t))V(x,t)$
      end
    end
  end
  return V

If π is stationary (not a function of time) [1] then as $t \to \infty$ the value function converges to fixed point satisfying the following *Bellman Equation*:

$$V^\pi(x,t) \xrightarrow{t\to\infty} V^\pi(x) = c(x,\pi(x)) + \gamma \sum_{x'} p(x'|x,\pi(x))V^\pi(x')$$

[1] Notably under assumptions that ensure convergence of the value function. If $\gamma < 1$ or, if, with probability 1, π enters a terminal state having zero cost.

Note that this equation is *linear* in $V^\pi(x)$. While this can be solved via policy iteration, an *alternate* way to compute this is to solve a system of linear equations.

Let $\overrightarrow{c^\pi}$ and $\overrightarrow{V^\pi}$ be vectors of length $|\mathbb{X}|$ listing the cost and cost-to-go, respectively for $\forall x \in \mathbb{X}$.

$$\overrightarrow{V^\pi} = \overrightarrow{c} + \gamma P^\pi \overrightarrow{V^\pi} \tag{5.0.1}$$

$$\Rightarrow (I - \gamma P^\pi)\, \overrightarrow{V^\pi} = \overrightarrow{c} \tag{5.0.2}$$

where P^π is the row stochastic transition matrix (its rows sum to 1) given the the fixed policy π

$$P^\pi = \begin{pmatrix} p(x_0|x_0,\pi(x_0)) & p(x_1|x_0,\pi(x_0)) & \dots \\ \vdots & \vdots & \vdots \\ p(x_0|x_n,\pi(x_n)) & p(x_1|x_n,\pi(x_n)) & \dots \end{pmatrix} \tag{5.0.3}$$

The operation of multiplying by P^π is the equivalent of calculating expectation. This is a linear equation in $\overrightarrow{V^\pi}$ and its solution is

$$\overrightarrow{V^\pi} = (I - \gamma P^\pi)^{-1} \overrightarrow{c} \tag{5.0.4}$$

For $\gamma < 1$ this equation always has a solution (the eigenvalues of P^π have modulus always less than one, so $I - \gamma P^\pi$ is always invertible).

Policy Improvement

If someone hands you a policy π, it is natural to want to see if it is optimal, and if not, to improve it (see Figure 5.0.1). Then, the question becomes, how can we tell that whether the policy is optimal? And, more importantly, if the policy is not optimal, "how can we modify the policy so that it becomes better, and eventually, optimal?" (See Figure 5.0.1) *Policy improvement* seeks to answer these questions.

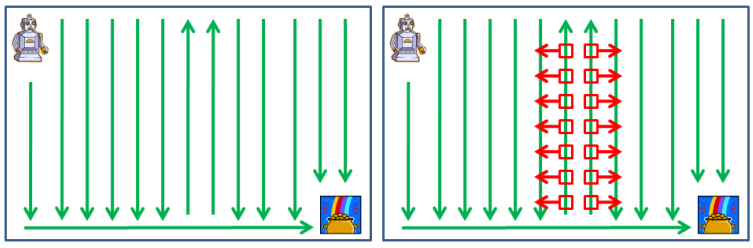


Figure 5.0.1: The left image shows a non-optimal policy (green arrows). The right image shows how the policy could be improved by changing the action taken on a state-by-state basis. The new policy is still not optimal and could be improved by another round of policy iteration.

In the *policy improvement* step, the policy is modified as follows $\forall x \in \mathbb{X}$:

$$\pi'(x) = \underset{a}{\operatorname{argmin}}\, c(x,a) + \gamma\, \mathbb{E}_{p(x'|x,a)}[V^\pi(x')]. \tag{5.0.5}$$

We can show that the new policy π' given by (5.0.5) will be at least as good as π. Moreover, as we shall see later, if the resulting policy π' is the same as the current policy π, then π is the already the optimal policy. Put differently, we only have an optimal policy if there is no change at any single state we can make that would appear to incur less long term cost.

Policy improvement can also be expressed in terms of $Q^\pi(x,a)$, the *quality function*, sometimes called the *Q-function* or *action value function*. The Q-function $Q^\pi(x,a)$ is the sum of the cost of performing an action a at state x and the expected cost to go from the resulting state under policy π.

$$Q^\pi(x,a) = c(x,a) + \gamma \, \mathbb{E}_{p(x'|x,a)}[V^\pi(x')] \tag{5.0.6}$$

A new policy π' can, therefore, be formed from an existing policy π by tweaking the action selected at a state. According to the policy improvement step, if π' is selected such that

$$\pi'(x) = \operatorname*{argmin}_a Q^\pi(x,a). \tag{5.0.7}$$

Policy Iteration Algorithm

Combining policy evaluation and policy improvement, we can get an algorithm, *Policy Iteration*, for finding a good policy from an arbitrary initial policy π_0.

Algorithm 9: Policy Iteration. Here the policy evaluation step can be computed by, e.g. Algorithm 8 or solving a linear system.

Start with arbitrary π_0
$k \leftarrow 0$
while *not converged* **do**
  Policy Evaluation: compute V^{π_k}
  for $\forall x \in \mathbb{X}$ **do**
    $\pi_{k+1}(x) = \operatorname*{argmin}_a c(x,a) + \gamma \sum_{x' \in \mathbb{X}} p(x'|x,a) \, V^{\pi_k}(x')$
  $k \leftarrow k+1$
return $\pi_k(x), \forall x$

5.1 *Policy Iteration Optimality*

During the policy iteration, the difference in value of the current policy π and the optimal value function $|V^\pi(x) - V^*(x)|$, decreases *exponentially* as a function of number of iterations. In practice– although with very little theoretical justification– it is found that policy iteration generally requires fewer iterations than Value Iteration. However, it does require more work on each iteration.

Understanding whether Policy Iteration will converge to the best policy is not trivial. The standard argument, outlined below, uses contradiction to show that there are no local optima, so, since each step is an improvement, the algorithm will converge to the optimum. To see this we need to show that:

- Policy Iteration monotonically improves
- Policy Iteration only produces no change in policy if it is at a global optima.

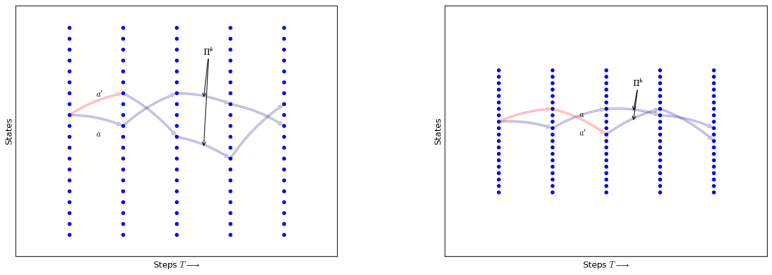
Together these imply reaching a global optima in finite time if there are a finite number of policies being considered. [2]

[2] Interestingly, it can be shown PI can theoretically visit an exponentially large set of policies, however, the policies distance from optimality decays geometrically in the number of iterations.

Monotonic Improvement

To show that the algorithm monotonically improves, we look at the improvement in the value function between policies. We switch actions *only if* (see Figure 5.1.1) the policy from that point onwards is an improvement.

To calculate the policy is improved, let us consider the difference between the value functions under two policies π and π' for a given initial state x_0. As is shown in Figure 5.1.1, let us consider the "value improvement" time-step by time-step.



(a) Choosing action a' in state x_0 which minimizes (5.0.5) at x_0, then following π

(b) Choosing action a' in state x_1 which minimizes (5.0.5), then following π

Figure 5.1.1: Improvement in the value function: Blue dots denote states, red arrows denote actions that minimizes (5.0.5) for a state, blue arrows denote actions in π.

The value improvement accrued at the initial time-step, due to the fact that π' chooses the action that minimizes (5.0.5) at x_0, must be:

$$\begin{aligned} & c(x_0, \pi'(x_0)) + \gamma \, \mathbb{E}_{p(x'|x_0,\pi'(x_0))}[V^\pi(x')] - V^\pi(x_0) \\ =& Q^\pi(x_0, \pi'(x_0)) - V^\pi(x_0). \end{aligned}$$

The value improvement accrued at the second time step (due to the fact that π' chooses the action that minimizes (5.0.5) at x_1) is:

$$\begin{aligned} & \mathbb{E}_{x_1 \sim p_1}\left[c(x_1, \pi'(x_1)) + \gamma \, \mathbb{E}_{p(x'|x_1,\pi'(x_1))}[V^\pi(x')] - V^\pi(x_1)\right] \\ =& \mathbb{E}_{x_1 \sim p_1}\left[Q^\pi(x_1, \pi'(x_1)) - V^\pi(x_1)\right], \end{aligned}$$

where $p_1(x_1) = p(x_1|x_0, \pi'(x_0))$.

Let's denote by $p_t(x)$ the probability of visiting state x at time t when we start at state x_0 and follow policy π', i.e. $p_t = Pr[x_t = x | x_o, \pi']$. By proceeding inductively from the above, we see that the difference between value functions can be calculated using the following equality:

Lemma 1. *Performance Difference Lemma:* [3]

$$V^{\pi'}(x_0) - V^\pi(x_0) = \sum_{t=0}^{\infty} \gamma^t \, \mathbb{E}_{x \sim p_t}\left[Q^\pi(x, \pi'(x)) - V^\pi(x)\right]$$

By the policy improvement step, we know that $Q^\pi(x, \pi'(x)) - V^\pi(x) \leq 0$ for all $x \in \mathbb{X}$. This lemma implies $V^{\pi'}(x_0) - V^\pi(x_0) \leq 0$ holds uniformly over all initial states and thus we see that the policy iteration algorithm improves the policy monotonically. [4]

[3] J. A. Bagnell, A. Y. Ng, S. Kakade, and J. Schneider. Policy search by dynamic programming. In *Advances in Neural Information Processing Systems*, 2003

[4] Assuming everywhere the infinite horizon value function exists, which of course holds for $\gamma < 1$

The Performance Difference Lemma is a powerful tool. It's proof, and the proof that policy improvement works in the Policy Iteration algorithm, have the same essential character.

Optimality

When policy iteration has stopped making improvements, i.e. a local optimum is reached,

$$(\pi, V^{\pi}) = (\pi', V^{\pi'}).$$

In this case, we have,

$$V^{\pi}(x) = V^{\pi'}(x) = \min_{a} c(x,a) + \gamma \, \mathbb{E}_{p(x'|x,a)} \left[V^{\pi}(x') \right].$$

Note this immediately implies that (π, V^{π}) are a solution to the Bellman Equation. Therefore $\pi' = \pi^*$ since (π^*, V^*) since the optimal value function– the Bellman Equation– is unique.

5.2 *Implementation Notes*

Often a *Modified Policy Iteration* is used in practice. Modified Policy Iteration warm-starts the policy evaluation step with the value function from the previous step and then does a few iterations k of policy evaluation. In the special case of $k = 1$, it reduces to VI (see Sutton and Barto Chapter 4). Since the expensive part of policy iteration is the policy evaluation step, this warm-start can greatly speed up the algorithm.

Dynamic programming algorithms (Value Iteration, Policy Iteration, Modified Policy Iteration, etc) are expensive if the state space is large. It can be used in its closed form (solving a linear system) if the value function is sparse. Otherwise the value function can be approximated by:

- Linear function approximator $\tilde{V}_{\theta}(x) = \theta^{\top} \phi(x)$, where $\phi(x)$ is the *feature* of the state x.
- Nearest Neighbour – for any x find the closest x' (in the sampled space) and return that value.
- Neural Network – a popular choice that has led to state of the art results in games from Backgammon to Chess.
- Any other regression algorithm....

Approximating the value function is the basis for the *Fitted Value Iteration* algorithm which we discuss in later lectures.

5.3 *Related Reading*

[1] Bagnell, J. A. , Kakade, S. Ng, A., Schneider, J. Policy Search by Dynamic Programming, NIPS, 2003.

[2] Puterman, M. Markov Decision Processes: Discrete Stochastic Dynamic Programming, 2005.

6
An Invitation to Imitation

Introduction

We take a detour now to study a conceptually simpler problem: that of *imitation learning*. Imitation learning is the study of algorithms that improve performance in making decisions by observing demonstrations from a teacher. Consider, for instance, Figure 6.0.1, which shows a human expert tele-operating a walking robot by commanding its footstep motions. Such motions and the decisions behind them are complex and difficult to encode in simple, manually programmed rules. While demonstrating a desired behavior may be easy, designing a system that behaves this way is often difficult, time consuming, and ultimately expensive. Machine learning promises to enable "programming by demonstration" for developing high-performance robotic systems.

Figure 6.0.1: Human expert demonstration to train a walking robot to cross very rough terrain. *Learning to Search (LEARCH)* [Zucker et al., 2011, Ratliff, 2009] attempts to make a footstep planner mimic the human pilot's choices. Imitation learning is the study of algorithms that improve decision making through data collected by observing an expert – often, but not always a person who can accomplish a task that is hard to hand-program.

Learning Behavior Without Generalization

Many of the references in imitation learning focus on learning fixed trajectories, or on controllers to achieve such trajectories in the presence of disturbances. (See a detailed discussion in [Argall et al., 2009, Osa et al., 2018].) Such work – including the foundational [Atkeson and Schaal, 1997] and the stunning helicopter acrobatics of [Coates et al., 2009] – vividly dramatizes the remarkable power of human demonstration. However, these approaches are limited in their ability to generalize to new circumstances. Our focus here is on strategies that can generalize to unfamiliar settings and base decisions on perceptual feedback. It is important to appreciate, however, that the boundary between trajectory learning approaches and general imitation learning is not clear. Atkeson [Atkeson and Morimoto, 2003], and others, notably [Safonova and Hodgins, 2007, Mülling et al., 2013], show that a library of trajectories can indeed be made to generalize very broadly through clever arbitration and blending.

Imitation ≠ Supervised Learning – The Distinctions

Unfortunately, many approaches that utilize the classical tools of supervised learning fail to meet the needs of imitation learning. We must address two critical departures from classical supervised learning to enable effective imitation learning.

Perhaps foremost, classical supervised machine learning exists in a vacuum. Predictions made by these algorithms are explicitly assumed to **have no effect** on the world in which they operate. We will consider the problems that result from ignoring the effect of actions that influence the world and highlight simple "reduction-based" approaches that mitigate these problems both in theory and in practice.

Second, robotic systems are typically built atop sophisticated planning algorithms that efficiently reason far into the future. Ignoring these planning algorithms in lieu of a reactive learning approach often leads to poor, myopic performance. While planners have demonstrated dramatic success in applications ranging from legged locomotion to outdoor unstructured navigation, such algorithms rely on fully specified cost functions that map sensor readings and environment models to a scalar cost. These cost functions are usually manually designed and hand programmed, which is difficult and time-consuming. Recently, a set of techniques for learning these functions from human demonstration by applying an Inverse Optimal Control (IOC) approach to find a cost function for which planned behavior mimics an expert's demonstration have been shown to be effective and efficient. These approaches shed new light on the intimate connections between probabilistic inference and optimal control. [1]

These two points are taken up in turn in the next two major sections.

6.1 *Cascading Errors and Imitation Learning*

Dean Pomerleau's work [2] on learning autonomous driving is the seminal work in the field of imitation learning. Moreover, it gets right to the heart of the differences between imitation learning and classical supervised learning. Figure 6.1.1 demonstrates the setup of Pomerleau's experiments on learning

[1] We prefer the older, more widely used, terminology *Inverse Optimal Control* as opposed to *Inverse Reinforcement Learning* (IRL) throughout. The central premise of research in inverse optimal control approaches to imitation learning is that the policy to be learned by demonstration can be thought of as a near-optimal policy for some plant with an unknown reward function. In Reinforcement Learning, by contrast, the plant itself is viewed as unknown. Thus we are typically solving the inverse problem of optimal control, but not of the inverse of reinforcement learning, rendering the phrasing IRL somewhat misleading. Moreover, it's valuable to connect to the original literature in control theory dating back to Kalman's [Kalman, 1964] foundational work.

[2] D. Pomerleau. ALVINN: An Autonomous Land Vehicle in a Neural Network. In *Advances in Neural Information Processing Systems (NIPS)*, 1989

to drive the NavLab vehicle by using a neural network to map camera images to steering angles. Pomerleau developed this procedure by driving the car and collecting pairs of coarse camera images and steering angles. He then trained a simple neural network in real time to take new images and predict the resulting steering angle. [3]

[3] The Pomerleau works truly hold up for today's reader both for their impact on autonomous vehicles and their deep insight into the key differences between supervised and imitation learning.

Figure 6.1.1: Pomerleau's Autonomous Land Vehicle in a Neural Network system at work driving the Carnegie Mellon NavLab vehicle. Used with permission.

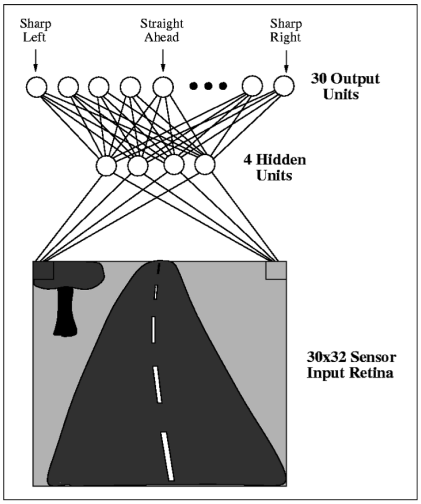


Figure 6.1.2: A schematic of Pomerleau's ALVINN driving system. The approach used a small neural network to map coarse camera images into a disretized set of steering angles. Image used with permission.

Consider a smaller, simplified version of the problem – learning to drive a car in a video game by performing a direct mapping from screen shots to steering angles. Figure 6.1.3 illustrates the classic supervised learning approach to learning such a mapping. [4]

Unfortunately, in this instance – as is quite common in practice – the approach fails disastrously and the learned controller quickly drives off the road. Let's consider what can go wrong. Of course, the learning problem

[4] Stephane Ross's results [Ross et al., 2011b, Ross, 2010a,b] applying such a procedure using linear regression on a simplified version of the screen image can be seen at Supervised Tux.

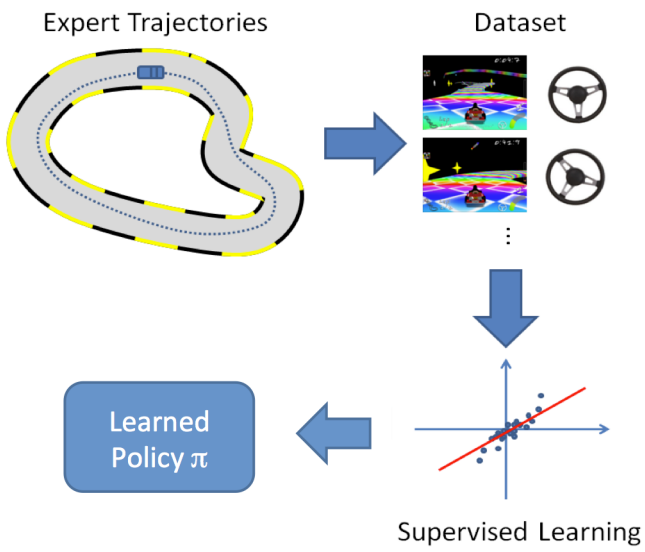


Figure 6.1.3: A sketch of the problem of learning to drive a video game simulation. A person drives the car around the course and collects data. That dataset consisting of images and associated steering angles is fed to a classic supervised learning algorithm, e.g., linear regression. The resulting policy π is used to drive the vehicle. Hilarity ensues.

may simply be too difficult. Perhaps we simply can't find a classifier or regressor that predicts the driver's steering decisions with small error. Perhaps a linear predictor is a bad choice for this problem; a richer hypothesis class might be more useful. That turns out not to be the case – a linear predictor is perfectly adequate for the task.

We could simply be *overfitting* – perhaps our training data set is too small to produce a good solution, which can lead to poor test performance. Avoiding overfitting has long been one of the central concerns in the study of learning theory[Shalev-Shwartz and Ben-David, 2014]. However, hold-out errors [5] are quite close to training errors in this example. Moreover, the learned policy[6] fails to perform well even with a very large set of training data.

What goes wrong? In a nutshell, learning errors *cascade* in imitation learning but are independent in supervised learning. Consider, for instance, a discrete version of the problem that only predicts "steer left" or "steer right". Inevitably, our learning algorithm will make some error – let's say with small probability ϵ for a good learner – and steer differently than a human driver would. At that point, the car will no longer be driving down the center of the road and the resulting images will look qualitatively different then the bulk of those used for training. Imitation learning has difficulty with this situation. The learner has never encountered these images before. Since learners can only attempt to do well in expectation over a distribution of familiar examples, an unusual image may incur further error, often with a higher probability.

As a result, the controller driving the simulation will steer the car close to the edge of the road – a very rare occurrence in training – and the resulting decision will likely be quite poor. Often, the learned controller will drive off the road, failing completely at the task. [7]

More formally, we can consider an imitation learning problem of T sequential decisions [Ross et al., 2011a]. If we learn a classifier making ϵ errors in predicting a driver's decisions in expectation over the distribution of examples induced by the teacher, we would hope to make $T\epsilon$ mistakes over the

[5] One can measure and control overfitting by considering the performance of a learned predictor on data that is "held-out": that is, data not available to the learning algorithm to train its predictor.

[6] We use *policy* here to refer to any learned predictor that maps features to actions. For discrete actions, this is simply a classifier. The terminology is common to optimal control and reinforcement learning, but is sometimes off-putting for roboticists and experts in supervised learning.

[7] Pomerleau's techniques for addressing these issues are particularly instructive. These include synthetic data generation, the use of online learning, and the emphasis on hard examples. This approach effectively manages *covariate shifts* similar to those caused when a learner influences its own test distribution. [Bagnell, 2005].

sequence of decisions. Unfortunately, an early error may compound into a long sequence of mistakes. As a result, the best we can hope for is $O(T^2\epsilon)$ mistakes[Ross and Bagnell, 2010]. [8] From a statistical point of view, our training and test data sets are not drawn from the same distribution and thus the supervised learning assumption of independent and identically distributed *(i.i.d.)* data is badly violated.

A natural suggestion for solving this problem is to collect data for all possible road conditions or over all images we may see. Unfortunately, it's difficult to obtain data for all possible inputs – the set of potential images is very large. Worse, no learner in our hypothesis class may be capable of handling all possible inputs. Assuming *realizability* – the "true" target function in our class– is generally far too strict, and algorithms that require this generally perform poorly. [Shalev-Shwartz and Ben-David, 2014] Instead, in machine learning we hope that there is a function in our hypothesis class that can work well *on average* over the actual distribution of training data that we encounter. [9]

[8] It's simplest to imagine a fixed time horizon. This fixed T can be replaced in analysis by notions of *mixing time*, *discount factor*, or a notion of how long any one mistake can propagate. It's therefore useful to consider T as representing an appropriate notion of the effective time horizon of the problem, not the actual number of decisions to be made.

[9] This point represents a general tension between the techniques of analysis in decision making and control – where one [Ljung, 1978] often requires a model or a controller to be *uniformly* good for all possible inputs, versus the paradigm of learning and statistics where it is recognized that this is not possible in high dimensional problems. In control, the focus is on ensuring good *expected* or average performance over the distribution of examples that actually occur. This mismatch lies at the heart of many of the difficulties of marrying learning and control. The interactive method discussed here – and no-regret learning in general – may serve as the bridge between these approaches.

A Simple Fix

If training data is plentiful and the time horizon is fixed and short, the compounding of errors is easily addressed. To proceed, we can train a policy for each of the T steps. The first policy is simply trained in normal supervised learning fashion by collecting data: the camera image and the person's steering angle at the initial decision. We train the next policy by executing the initially learned policy for the first time step, then turning over the wheel to the teacher. A new data set is collected for the second time step, consisting of the input images seen by the teacher at time 2, and the resulting steering decisions. A policy can then be learned for time step 2 via the usual machinery of supervised learning. We can easily repeat this process to train the k-th step in a time-varying policy by observing the teacher's decisions after running the first $k-1$ steps of the learned policy [Ross and Bagnell, 2010].

It follows that each policy learned is being tested in exactly the way it was trained. The policy encounters the same distribution of input examples– albeit not the same actual examples! If an earlier policy makes errors, later ones can learn to recover from them by mimicking the teacher's recovery strategy. This halts error compounding and achieves the error rate $T\epsilon$ that one would expect in standard supervised learning.

A practical solution: DAGGER

While the above approach cleanly addresses the problem of decisions affecting the input distribution in imitation learning, it is impractical for imitation learning problems like the video game driving problem. We simply can't afford to train a policy for every step in a long sequence of decisions like driving a vehicle. Moreover, this process **should** be unnecessary if the effective time horizon is shorter.

A solution to this problem relies on *interaction*: interleaving execution and learning. In particular, at each iteration of the algorithm, the current learned policy is executed. Throughout execution, the teacher "corrects" the solution – that is, provides a preferred steering angle that is recorded in a new data set but **not** executed. After sufficient data is collected, it is

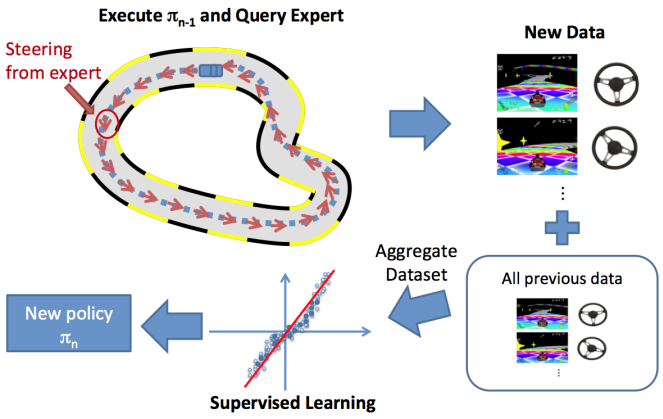


Figure 6.1.4: Illustration of the Dataset Aggregation (DAGGER) approach to imitation learning via repeated interaction. At each iteration of the algorithm, the current learned policy is executed. Throughout execution, the teacher "corrects" each step – that is, provides a preferred steering angle that is recorded in a new data set but **not** executed. Throughout these iterations, data is aggregated together to lead to the next policy. This provides much stronger guarantees than simple supervised learning.

aggregated together with all of the data that was previously collected. A supervised learning algorithm then generates a new policy by attempting to optimize performance on the aggregated data. This process of execution of the current policy, correction by the teacher, and data aggregation and training is repeated.

```
# Take an initial policy: π_0, Teacher: state -> action,
# Learner: [ (state,action) ] -> policy, GenSystemTrajectory : π -> [state]
def DAGGER(π_0, Teacher, GenSystemTrajectory, Learn):
    D = [], π = π_0
    for i in range(N): # run for N iterations
        D_i = [(state,Teacher(state)) for state in GenSystemTrajectory(π) ]
        D.append(D_i)
        π = Learn(D) #Optionally run any no-regret learner on the D_i
    return π
    # Preferred: instead return the stochastic policy that mixes uniformly between all the
    # policies learned or choose the best single policy on validation over the iterations
```

DAGGER Algorithm Pseudo-code

Intuitively, this approach creates policies that are capable of correcting their own mistakes. If the learner steers too close to the edge of the road, the policy will generate new training data that includes the teacher's preferred actions for handling such situations. The aggregation of data prevents it from forgetting previously-learned situations.

But what can one say formally about this approach? If our supervised learner is one of a large class of learners that have the *no-regret* property[Cesa-Bianchi et al., 1997], we can formalize the idea that learning a policy with low training error implies good performance at imitating the expert. Put differently, one of two things must happen: either the supervised learning problem will become too hard to solve (expected error greater then ϵ) or a policy that matches the teacher with only approximately $T\epsilon$ error over the full horizon will be learned throughout the iterations. [10]

[10] It need **not**, however, be the final policy learned. Instead, the claim is merely that one of the policies– or a uniform stochastic mixture of the entire set learned– must perform well. In practice, choosing the final learned policy is often simplest and sufficient.

Stability, Online Learning and "No-regret"

Case Study: DAGGER *in Anger*

When we apply this approach of teacher correction, aggregating data and iteratively learning policies to the car driving problem, the result is somewhat boring to watch. While simple supervised learning averages about 3-4 failures per lap, the interactive DAGGER learning approach **with the same number of examples from the teacher** very quickly reaches nearly 0 falls per

lap. **No** amount of training data enables the supervised learning approach to achieve that same performance– it always falls multiple times per lap.

It's more interesting to consider learning a complex, real-world reactive control task like flying through a cluttered domain – for example, between tree trunks underneath a forest canopy. [11] The problem follows the setup of Pomerleau's: compute features (optical flow, color histograms, simple texture features etc.), pool them over patches of the images, and provide the resulting large feature vector as an input to a regression algorithm. As output, the learner will predict the commanded lateral velocity of a human pilot and train the algorithm to reactively map these image features to controls.

[11] The "Forest of Endor" problem, to use Nick Roy's evocative phrase.

The result is a simple controller that navigates through dense forest at nearly the same effectiveness as a human pilot. [Ross et al., 2013a] [12]. Interestingly, failures largely come about due to the nature of a reactive controller and a small field of view. It's not unusual for the algorithm to dodge a tree, have that tree leave its field of view, then crash into the same tree sideways as it tries to avoid a new tree. Adding memory – whether through intelligently constructed features or through predictive state representations – represents the best hope for improving the learning of such control strategies.

[12] Videos of the approach can be found at LAIRLab BIRD Website [Ross et al., 2013b]

Recently other authors have demonstrated in success in applying DAGGER to a rich class of problems including playing a broad class of Atari 2600 games [Guo et al., 2014] and robot navigation [Kim et al., 2013].

Learning with a Goal Besides Imitation We focused entirely above on a loss function of **simple imitation**: our goal is to choose the same actions as the expert measured according to some loss function $l(y, \pi(x))$. But in many scenarios – for instance, driving – our real goal is actually substantially different. We may wish to minimize the probability of crashing, or maximize our success at manipulating an object, or achieve any other control objective that the teacher is presumably optimizing. The same style of approach is easily adopted – albeit with potentially substantially higher computational and sample complexity – for this setting by replacing the data about best action with an estimate of cost-to-go from the teacher. [Ross and Bagnell, 2014] Crudely speaking, this cost-to-go is an estimate of how hard it will be for the teacher to recover if the learner were to make a mistake. The key question of what to do when a teacher can't articulate their own cost function is taken up in the next section.

Summary

In an important sense, recent theory and algorithms for imitation learning formalize a simple lesson: one cannot learn to drive a car simply by watching someone else do it. Instead, feedback is essential – we must try to drive and receive instruction that corrects our mistakes.

Crucially, this general approach is largely agnostic to the underlying supervised learning approach. It is an interactive *reduction* to supervised learning methods. Formal results are only known for settings (like kernel machines, Gaussian processes, and linear predictors) where no-regret algorithms are known. But empirical evidence suggests that this approach is remarkably effective even when this condition doesn't formally hold, since many learning algorithms are actually both stable and good predictors.

Finally, it is important to note that all discussion here centered on learning

mappings directly from observations to controls without considering state-estimators (e.g. *filters*.) However, there is no reason one can not nor should not learn to *imitate in belief space*– that is learn mapping from the output of a filter (*e.g.* a best estimate of the underlying world state) to decisions. In practice, this is almost certainly necessary to achieve high performance; such approaches fall under the same general approach described here as we can consider the filter as simply a part of the environment and the filter output as a new, generalized observation.

6.2 *Decisions are Purposeful: Inverse Optimal Control*

Figure 6.2.1: An image of the DARPA UPI "Crusher" robot autonomously crossing rough off-road terrain. It is difficult to manually engineer the connection between perception and planning. Imitation learning techniques make it possible to automate this process. Further examples of the vehicle traversing rough terrain from temperate woodlands, to marshes, to dense vegetation, to mock-up urban environments all under autonomous control can be seen here and here.

Imitation learning is fundamentally different then classical supervised learning in another sense. For instance, consider the problem of navigating through very rough outdoor terrain – a major focus of robotics research for decades. Figure 6.2.1 shows Crusher, an autonomous robot that was developed as part of a DARPA fundamental research project into outdoor robotics. Crusher traversed thousands of kilometers of diverse, rough, terrain with minimal human intervention over years of field testing. In contrast to many other outdoor navigation efforts, it typically travelled from 0.5 to 10 kilometers between human provided waypoints. All decisions along the way were made based on information from its own perception system and (optionally) overhead maps (*e.g.* images collected from mapping companies like those used in Google Maps).

A reactive controller is unlikely to make any meaningful progress towards a goal in this domain; it is difficult to imagine training a simple supervised learning method to accomplish this complex task. The robot must instead execute a long, *coherent* sequence of decisions in order to achieve its goal. This requires a sense of planning – and of replanning as new perceptual information becomes available – to achieve good performance.

To adapt to imitation learning to this setting, it is valuable to consider the architectures that roboticists have created to achieve intelligent and deliberative navigation. Since the pioneering projects in off-road navigation [Hebert, 1997], effective robot navigation has relied on an optimal control or replanning architecture to structure decision making. This architecture has been replicated and refined throughout the field of robotics [Zucker et al.,

2011, Urmson et al., 2008, Wellington and Stentz, 2004, Leonard et al., 2008, Jackel et al., 2006, Bachrach et al., 2009] and is currently used in the most advanced autonomous navigation systems.

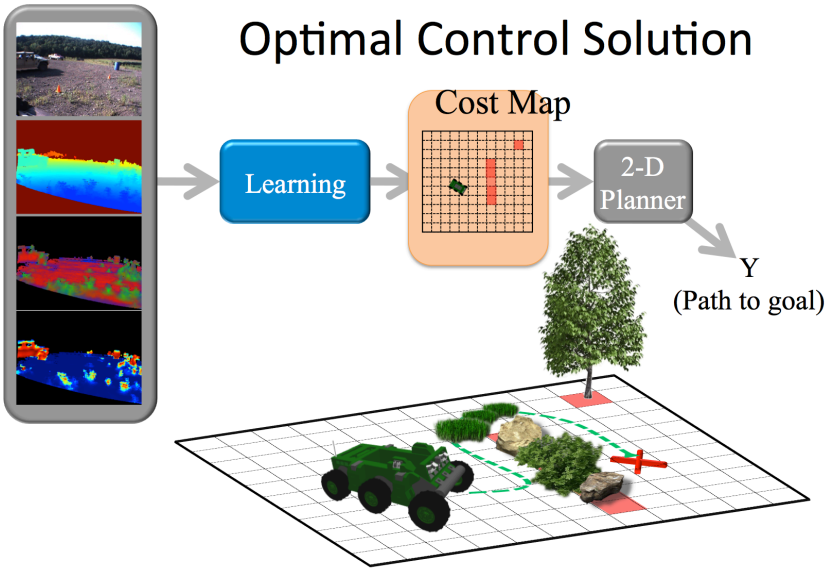


Figure 6.2.2: Components of a robot architecture: Sensors (LADAR, cameras) feed a perception system that computes a rich set of features (left side) developed in the computer vision and robotics fields. Depicted features include estimates of color and texture, estimated depth, and shape descriptors of a LADAR point cloud. Features that are not depicted here include estimates of terrain slope, semantic labels ("rock"), and other feature descriptors that can be assigned a location in a 2D grid map. These features are then massaged into an estimate of "traversability" – a scalar value that indicates how difficult it is for the robot to travel across the location on the map.

Figure 6.2.2 shows a diagram of such a robot architecture. Sensors (LADAR, cameras) feed a perception system that computes a rich set of features (left side) developed in the computer vision and robotics fields. Features that are shown in Figure 6.2.2 include color, texture, estimated depth, and shape descriptors of a LADAR point cloud. Features that aren't shown in the diagram include estimates of terrain slope, presence of semantic categories ("rock"), and many other feature descriptors that can be assigned a location in a 2D grid map. These features are then massaged into an estimate of "traversability" – a single scalar value that that indicates how difficult it is for the robot to travel across the location on the map. This value is included in a "cost map" for each state of the robot. The final decisions of the robot represent steps along a minimum cost plan from the robot's current location to a goal state. The robot executes a small part of the current plan at each time instant. As the robot moves, the perception system provides updates about the terrain it is crossing. The cost map is then updated with new traversibility values and a new plan is generated.

Real implementations, of course, have much richer spaces of states then simply a discretization of geometric locations of the robot center. Almost inevitably, they contain a hierarchy of planning layers that capture a state-space description of the robot at higher and higher fidelities as they consider shorter time-scales. [Zucker et al., 2011] The diagram in Figure 6.2.2 nevertheless captures the essential behavior of many such systems and is often exactly the behavior of the coarsest levels of such a hierarchy.

From the point of view of this architecture, only one role exists for imitation learning. Perception computes features that describe the environment;

the output control is always the prefix of the currently believed-to-be-optimal plan. The learning algorithm then must transform the perceptual description (a feature vector) of each state into a scalar cost value that the robot's planner uses to compute optimal trajectories.

Perhaps surprisingly, costing is one of the most difficult tasks in autonomous navigation. As documented in [Silver, 2010], this single piece of code required the largest number of changes and demanded the most engineering effort. The entire behavior of the robot depends on this module working correctly. Moreover, nearly all changes to the software end up requiring either validation or modification of the costing infrastructure. If a sensor changes or the perception system develops or refines features, the costing mechanism must be updated. If the planner changes – for instance by C-space expanding obstacles– the costing system must change. Tuning and validating such changes demands a tremendous amount of time and effort.

However, the robot can use imitation to learn this cost-function mapping. A teacher (that is, a human expert driver) drives the robot between waypoints through a representative stretch of complex terrain. We can then set up a problem of *Inverse Optimal Control*: that is, we attempt to find a cost function that maps perception features to a scalar cost signal so that the teacher's driving pattern appears to be optimal.

Nathan Ratliff formulated the problem of learning such a cost function as an application of *structured prediction* and demonstrated that very simple sub-gradient based algorithms are remarkably effective at solving it. [13]

Inverse Optimal Control (IOC) is a rich and fascinating subject that dates back to Kalman's work on the Linear-Quadratic-Regulator problem. Kalman [Kalman, 1964] asked (and answered) a natural question: given a linear controller or policy, is there a cost function that makes it optimal for a given Single-Input Single-Output plant?[14] Boyd [Boyd et al., 1994] provided a simple convex programming formulation for the multi-input, multi-output linear-quadratic problem.

Only recently, however, has Inverse Optimal Control become an engineering tool for designing intelligent systems. The recent work in the machine learning on this area [Ng and Russell, 2000, Abbeel and Ng, 2004, Ratliff et al., 2009c, Ziebart et al., 2008a, 2010] can be summarized as providing several advances over the early contributions:

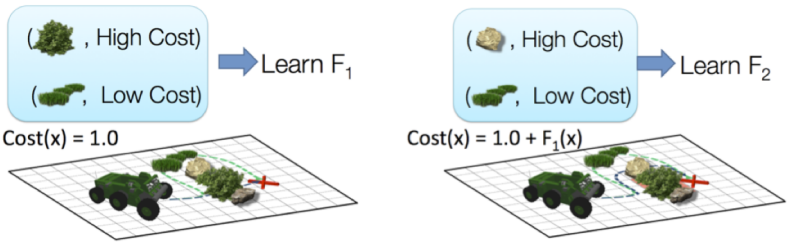


Figure 6.2.3: Iterations of the LEARCH algorithm. See the main text for a description of how this algorithm modifies its estimate of a cost function by mapping features of a state to a scalar traversability score.

(1) Enabling a cost function to be derived (at least in principle) for essentially arbitrary stochastic control problems using convex optimization techniques – any problem that can be formulated as a Markov Decision Problem.

(2) Requiring a weak notion of access to the purported optimal controller.

[13] In fact, surprisingly such sub-gradient methods are actually the best known algorithms for solving large support vector machine and more general structured margin problems in a follow-on paper. These techniques are now the de facto standard and have been implemented in a wide range of libraries [Agarwal et al., 2014].

[14] Amusingly, while Kalman's work was critical in advancing the use of state-space techniques for control, his solution to the IOC problem was rooted fundamentally in frequency domain techniques.

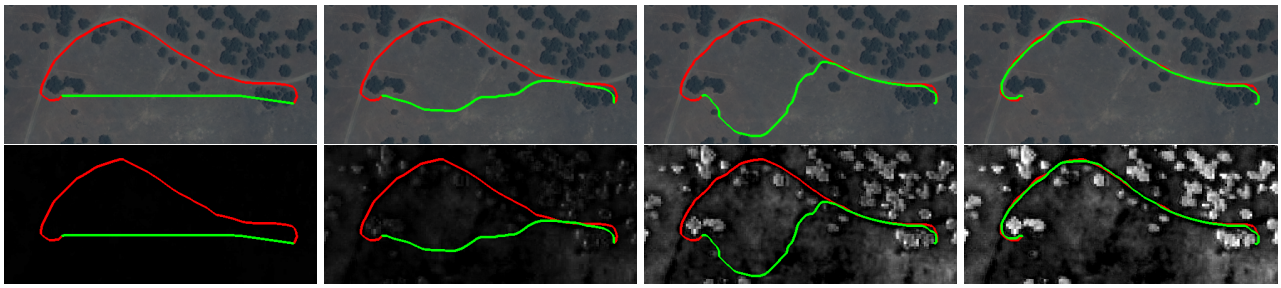


Figure 6.2.4: A demonstration of the Learning to Search (LEARCH) algorithm applied to provide automated interpretation in traversability cost (Bottom) of satellite imagery (Top) for use in outdoor navigation. Brighter pixels indicate a higher traversability cost on a logarithmic scale. From left to right illustrates progression of the algorithm, where we see the current optimal plan (green) progressively captures more of the demonstration (red) correctly.

No closed form description of the controller needs to exist, just access to example demonstrations.

(3) Statistical guarantees on the number of samples required to achieve good predictive performance and even stronger results in the online or no-regret setting that requires no probabilistic assumptions at all.

(4) Robustness to imperfect or near-optimal behavior and generalizations to probabilistically predict the behavior of such approximately optimal agents.

(5) Some algorithms further require **only** access to an oracle that can solve the optimal control problem with a proposed cost function a modest number of times to address the inverse problem.

The central premise of IOC techniques for imitation learning is that structuring a space of policies as approximately optimal solutions to a control problem is a representation that enables effective deliberative action. Moreover, IOC methods rely on the observation that cost functions generalize more broadly [Ng and Russell, 2000] then policies or value functions. Thus, one should seek to learn and then plan with cost functions when possible, and revert to directly learning values or policies only when it is too computationally difficult.

The Learning To Search (LEARCH) Algorithm. The key algorithmic ideas for modern IOC algorithms statistical guarantees can be understood in the framework of convex optimization of an objective function that stands as a surrogate for correctly predicting the plan or policy that the teacher will follow. As such, many of the original approaches were formulated in terms of large quadratic programs [Ratliff et al., 2006] or Linear-Matrix Inequalities [Boyd et al., 1994] and the resulting algorithms are somewhat opaque. However, more recent algorithms designed for solving large scale and non-linear problems are quite natural and might be guessed without even appreciating they are solving a well-defined optimization problem.

Consider, for instance, the *Learning to Search* (LEARCH) approach of [Ratliff et al., 2009c] in the context of rough-terrain outdoor navigation discussed above. We may step through the algorithm on a cartoon example to see why it might work. We first consider a path driven by teacher from a start point to a goal point, then imagine a simple planning problem on a discretized grid of states that the robot can occupy. Every iteration of the algorithm consists of the following: (a) computing the current best optimal plan/policy; (b) identifying where the plan and teacher disagree and creating a data set consisting of features and the direction in which we should modify

the costs; (c) using a supervised learning algorithm to turn that data set into a simple predictor of the direction to modify costs; and (d) computing a cost function as a (weighted) sum of the learned predictors.

```
# Take a sequence of MDPS and demonstrations [M_i,ξ_i)]_{i=1}^N where MDP M is a stochastic planning problems
      consisting of states, actions, and a transition function used for planning,
#  (optional) loss functions l_i: state, action->R that measures deviations from the demonstrated plan,
# feature function f: state, action -> R^d  that describes states in terms of features meaningful for
      cost

def LEARCH({(M_i,ξ_i)}_{i=1}^N, f, {l_i}_{i=1}^N = 0):
  s_0 = 0 #Initialize (log)-cost function, s_0 : R^d → R to zero
  for t in range(T): # run for T iterations
      D = [] # Initialize the data set to empty
      for i in range(N): # for each example in the data set
          c_i^l = e^{s_t(F_i)} - l_i^T # Compute costmap with optional loss augmentation
          μ_i^* = Plan(M_i,c_i) # find the resulting optimal plan μ_i^* = argmin_μ c_i^l μ, μ consistent with M_i
          # μ*'s counts the time spent in state/actions pairs under the plan—
          # for deterministic MDPS this is simply an indicator of whether the optimal plan
          # visits that edge in the planning graph
          μ_i = [ ξ_i.count((s,a)) for (s,a) in M_i ] #compute states-actions in demonstration
          # Generate positive and negative training examples:
      D_i = [ (f_i(s,a), sign(μ_i^{*sa} - μ_i^{sa}), |μ_i^{*sa} - μ_i^{sa}|) for (s,a) in M_i ]
      # if |μ_i^{*sa} - μ_i^{sa}| = 0 for a state-action we can simply not generate that point
      D.append(D_i)
      h_t = Learn(D) #Train a regressor (or classifier) h_t: R^d->R on the resulting weighted data set
      s_{t+1} = s_t + α_t h_t # Update the (log) hypothesis cost function
  return exp(s_T)
```

LEARCH Algorithm Pseudo-code

Theory and Guarantees. At its heart, the problem of correctly identifying a teacher's reward function is ill-posed. First, it is unreasonable to believe the teacher is truly an optimal controller for some simple Markov Decision Process that describes the world. Second, given a single behavior, there are generally **infinitely many** reward functions that lead to the same behavior and are thus unidentifiable. [Abbeel and Ng, 2004]

There are thus two commonly used notions of successful IOC used in machine learning. The first (originated by Abbeel and Ng [Abbeel and Ng, 2004]) considers a class of reward functions that are linear in a set of features that describe states. Our goal then is to ensure that whatever behavior is learned by imitation achieves the same reward as the teacher even when the reward function itself cannot be identified. The second (typified by *Maximum Margin Planning* [Ratliff et al., 2006, 2009c]) is agnostic to whether the teacher is actually an optimal controller or even cares about a reward function. Instead, it quantifies a notion of successful imitation – for instance, agreement with the trajectory taken by the teacher – and then attempts to optimize that notion of agreement with the teacher.

These notions are surprisingly closely tied. Methods like *Maximum Margin Planning* that ensure successful agnostic imitation also can provide guarantees with respect to the teacher's reward function (if it exists!).[Syed and Schapire, 2007] Conversely, while methods like the Maximum (Causal) Entropy approach of [Ziebart et al., 2008a], which we cover extensively in the next chapter, are also designed to achieve the same reward as a teacher, they can also be understood in a dual formulation as maximizing the likelihood of the teacher's plans under a robust statistical model of the agent's behavior. [Ziebart et al., 2010, 2013] Moreover, some methods, like that of [Ziebart et al., 2010], have yet another interpretation in terms of optimal control perturbed by certain shocks that are not visible to the learner. [Rust, 1994]

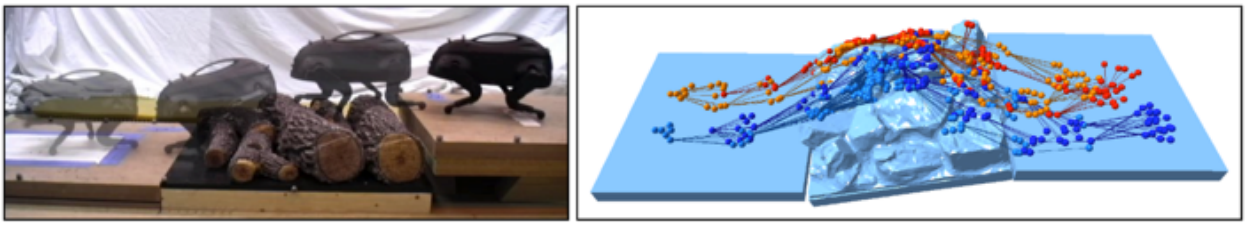


Figure 6.2.5: (Left) LittleDog platform crossing a terrain. (Right) Planning system that relies on a learning approach to cost function generation. Each color represents a different foot and arrows indicate the parent/child relationship between footsteps under consideration. [Zucker et al., 2011]

IOC in other Domains The notion of learning such deliberative strategies by tuning the cost function of a planner isn't unique to outdoor navigation– it arises anywhere long horizon plans are needed and relatively complicated features exist to describe the state space. [Ratliff et al., 2006, Zucker et al., 2011] developed a technique for learning costs (and a hierarchy of heuristics) by demonstration (see Figure 1) for a rough terrain legged locomotion planner. In essence, quasi-static locomotion is treated as discrete planning problem of carefully arranging footfalls. A complex cost function that takes into account the terrain at each individual foot as well as features of the entire robot pose that are correlated with good foot placements (for instance, the size of the polygon of support of the robot [Zucker et al., 2011]) was learned from expert demonstration. Multiple research groups have since embraced similar techniques [Kalakrishnan et al., 2011, Kolter et al., 2007].

Purposeful Prediction. Often, behavior demonstrated is only approximately optimal or may appear to have some non-determinism in its decisions. This can be understood in two ways: people are not in fact "optimal" in their decision-making for any reasonable definition of that word, and even more so, the world those people inhabit is not the simple Markov Decision Process we use as our model in Inverse Optimal Control techniques. [15] Recent IOC learning techniques manage such uncertainty and moreover can make probabilistic predictions of what people are likely to do even in such imperfect models.

The ability to imitate a person's imperfect but deliberative behavior implies the ability to predict it. In Figure 6.2.6 we see examples of Activity Forecasting: predicting people's likely trajectories in novel scenes via computer vision and inverse optimal control by learning what they are approximately optimizing in their decision making. [Kitani et al., 2012]

For instance, consider the problem of predicting the likely routes that a driver might take to travel from home to a store. We can consider a graph that describes the road network with nodes corresponding to road segments and edges between road segments that connect. Each road segment is annotated with a rich set of features x (dozens or hundreds) that describe it [Ziebart et al., 2008b] – such as expected travel times at the speed limit, the grade of the road, the toll cost of that segment, the number of lanes, whether a church is located along the road, or the presence of a guarded left turn.

The approach of [Ziebart et al., 2008a] efficiently learns a function $c(x)$ that linearly combines such features to best fit a distribution over trajectories ψ taken by the driver according to the maximum entropy model $p(\psi) \propto \exp(-V(\psi))$, where V is the total cost of the trajectory, $\sum_{x \in \psi} c(x)$.

[15] *I.e.,* the map is not the terrain.

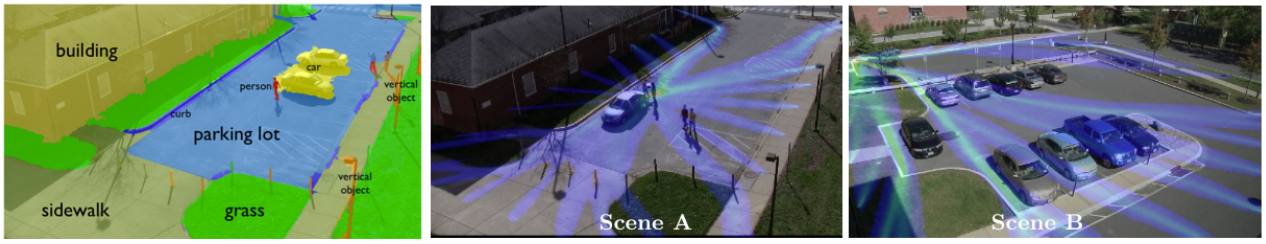


Figure 6.2.6: (Left) Automatic semantic classification of a scene via machine learning[Munoz et al., 2010, Munoz et al., 2013] (Right) Activity forecasting combines semantic perception techniques to identify the actors and object types in a scene with the probabilistic formulation of inverse optimal control to predict an actor's future destinations and likely trajectories based on partial trajectories. Each image depicts the predicted distribution of future states for a pedestrian. The absence of color implies very low probability, blue implies low probability, and yellow to red higher. Only a few potential goals are shown, and only with a single observation (predictions improve as more of the path is seen), to simplify the figure. [Kitani et al., 2012, Ziebart et al., 2013, 2008a]

When these models are combined with a prior distribution over potential destinations, they learn both a driver's implicit preferences (for example, going out of the way to avoid both unguarded left turns and expensive tolls) and provide a estimate of a drivers destination and likely future routes after beginning a trip. The use of the maximum entropy formulation ensures a strong guarantee on the predictions– no other approach to forecasting an agent's actions that uses the same information about features [Ziebart et al., 2013] can ensure smaller predictive loss.

This approach establishes the deep connection between probability theory, and particularly the Maximum Entropy Method, and inverse optimal control, where previously, these were understood as disparate techniques for modeling decision-making. [Ziebart et al., 2008a] This thread of work culminated in a new principle for the statistical prediction of interacting systems (e.g. a driver and the world, multiple agents playing a dynamic game) [Ziebart et al., 2010, 2013]. [16]

Similar techniques can be applied to predict where people are likely to walk in a complex visual scene. For instance, such methods could recognize cars and sidewalks in a scene and reason that a person will climb over a car if strictly necessary to reach a goal, but will preferentially take advantage of a sidewalk where available. [Kitani et al., 2012] Moreover, such techniques have been applied to aid robot navigation and predict pedestrian behavior. [Ziebart et al., 2009, Kretzschmar et al., 2014]

Work by [Baker et al., 2009] demonstrates people reason about others as deliberative agents as well. This inverse planning framework elegantly captures aspects of the human "Theory of Mind." Work in operations research and econometrics, particularly by Rust [Rust, 1992, 1994], derives predictive distributions by developing a framework for learning cost functions and predictive stochastic policies for agents acting according to a Markov Decision Process (MDP). Intriguingly, the same policy structure and dynamic programming algorithms derived from a maximum entropy formulation are developed from considering an economist with only partial access to the prediction problem and including "shocks" in a model of what would otherwise be optimal behavior. These close connections between operations research, control theory and machine learning deserve much deeper investigation.

6.3 *Structured Prediction as Imitation Learning*

At first blush, it seems counter-productive to phrase a supervised learning problem as one of imitation learning. Isn't the point of this article that imitation learning is a *harder* problem then that of supervised learning? However,

the relationship between the two is more subtle than this simple picture suggests. Within supervised learning, we often consider problems of *structured prediction* where the goal is to make a set of inter-related predictions – for instance, to semantically label all of the pixels within an image (e.g., Figure 6.2.6) or to turn a sentence into a parse tree. [Daumé III et al., 2009] suggests that a natural way to think about structured prediction is to consider it as predicting a sequence of decisions – *e.g.* what to label a particular pixel *given* current guesses of labels – and moreover that the expert we are imitating is simply the ground truth. [17]

From this viewpoint, structured prediction problems are merely degenerate versions of imitation learning problems, where the teacher can be specified algorithmically based on training data and the dynamics of the environment are particularly simple. When viewed through this lens, structured prediction problems suffer the same difficulties as problems of imitation learning. Predictions of some random variables (e.g., pixel classes) influence future predictions of other pixels and a naive training of such an architecture leads to disastrous compounding of errors.

For instance, consider the *inference machine* approach of [Munoz et al., 2010, Ross et al., 2011b]. The central idea is to consider labeling an image or point cloud sequentially in a pattern mimicking that of highly effective graphical model inference algorithms like *mean-field* or *belief-propagation*. We iteratively pass through each pixel and label it using a combination of (a) features that describe that particular visual element (e.g. texture, color) as well as (b) the currently predicted labels of visual elements that are nearby. The use of such nearby elements for predictions enables effective contextual reasoning. It's easier to distinguish a tree trunk from a telephone pole if we know that the material located above it is vegetation. Such contextual reasoning has traditionally been approached through the lens of probabilistic graphical models. We first learn a templated parameterized probabilistic model, then use approximate inference techniques to infer random variables in that model. The imitation learning approach makes the inference procedure itself the model. [18]

Such techniques— and more generally, applying methods like DAGGER to structured prediction– have been demonstrated to provide state-of-the-art predictive performance and speed of inference on a wide range of structured prediction tasks. These include examples from predicting semantic labels for images [Munoz et al., 2010], identifying human poses in images and video [Ramakrishna et al., 2014], summarizing documents with the SCP algorithm [Ross et al., 2013c], and a broad range of Natural Language Processing Tasks [Daumé III et al., 2009, He et al., 2012]. [19]

6.4 What's Next?

Only in the past decade has imitation learning come into its own as a problem distinct – and distinctly important – from the classical ones of reinforcement and supervised learning. The structure of the problem gives us far more purchase then the general reinforcement learning (RL) problem. But it also acknowledges that learning may actually affect the world and that the classic assumptions of supervised learning will lead to poor performance and compounding errors.

[17] Hal Daume at a NIPS workshop first clearly expressed to me the notion that we should often think of supervised learning problems as being imitation learning problems in disguise. This viewpoint has certainly been addressed by others – John Langford has particularly championed the notion that complex prediction problems should be thought of in terms of reductions to simpler problems.

[18] It is natural to view the inference machines in the language of deep modular neural networks [LeCun et al., 1998, Bengio, 2009] – an inference machine is a very deep set of repeated predictions about a particular visual element or other random variable. An alternative to the iterative training procedures espoused here includes a direct backpropagation of errors of final predictions made about such nodes. Interestingly, however, such results limit our prediction algorithms (no random forests!) and may not always be an optimal approach. Investigating when backpropagation can effectively tune the parameters of an inference machine remains an important subject for research.

Shih-En Wei, Varun Ramakrishna, Takeo Kanade, and Yaser Sheikh. Convolutional pose machines. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4724–4732, 2016

[19] Videos of such inference approaches approaches can be found at the Inference Machine Website.

Apprenticeship: From Imitation to Reinforcement

An important next step is moving from pure imitation to apprenticeship[20], which leverages user demonstration but optimizes performance on an alternate metric. Many examples in the literature consider where it can have significant benefits. For instance, [Nechyba and Bagnell, 1999] demonstrates a learned speed control for a simulated driving task that is improved by an RL gradient descent procedure that ensures good performance while attempting to stay as close as possible to demonstration. Similarly, the works of [Atkeson and Schaal, 1997], [Kober and Peters, 2010] and [Coates et al., 2009] use exactly same kind of benefits to achieve impressive performance. Such approaches are even more important when the learning **cannot** be interactive– for instance, when learning by watching a video.

[20] Borrowing this phrase from Pieter Abbeel, who uses it to refer to systems that combine imitation and reinforcement learning.

Interestingly, theoretical results suggest an enormous practical benefit for learning from an expert demonstrator – but perhaps not in the way typically considered. The theories of Policy Search by Dynamic Programming [**?**], Conservative Policy Iteration [Kakade and Langford, 2002], and No-Regret Policy Iteration [Ross and Bagnell, 2014] show that the key to making reinforcement learning easier is to identify the distribution of states that a good policy spends time in (the so-called *baseline measure* of [**?**]). Access to such a distribution makes the problem of a learning an optimal memory-less policy in a Partially Observed MDP a polynomial-time problem. It also effectively makes the sample complexity of learning into a policy with generative model access to a large MDP polynomial in the horizon of the problem.

Such results, however, show no significant benefit for observing what actions an expert demonstrator might choose – the benefit of this seems secondary to the benefit of knowing what states are important to focus on. Understanding practically and theoretically how we can get the best of imitation and reinforcement learning will be a major area of future research.

Extending Inverse Optimal Control for Imitation Learning.

Much recent work has focused on models for which the optimal control problem itself can only be approximately solved. [21]

Such methods and combinations of methods seem likely to dramatically increase the applicability of this rich class of predictive models and procedures for inferring reward functions.

[21] [Ziebart et al., 2012] and [Dragan and Srinivasa, 2012] and [Levine and Koltun, 2012] consider locally quadratic approximation of the maximum entropy inference problem. [Huang et al., 2015] has developed a variant of the maximum entropy IOC that relies on a combination of function approximation of the log-partition function and sampling to estimate the gradient. [Ratliff et al., 2009a] blends the advantages of IOC-based methods with methods that directly learn to predict actions.

Putting it together

Perhaps surprisingly, existing techniques rarely consider both aspects of imitation learning I have discussed in this paper: they tend to focus either on the problem of compounding errors or the need for learning deliberative strategies. As these problems are largely orthogonal, we expect future techniques for imitation learning will address both issues simultaneously.

7
Moment Matching, GANs, and all that

Introduction

In this chapter we review Inverse Optimal Control from the *Maximum Entropy* perspective and connect these to the general goal of learning probability distributions from examples. This chapter establishes the key role of both moment matching/integral probability metrics and a game theoretic view of learning behavior. This viewpoint allows us to connect IOC and the Maximum Entropy Principle more broadly to a family of generative models known as *Generative Adversarial Networks*. Efficiency is achieved for continuous control problems via a Laplace approximation and techniques are studied to learn costs and find anomalies with this approximation.

This chapter extends the previous one by considering methods for learning cost functions from human demonstration that highlight the intimate connections between probabilistic inference and optimal control. Moreover, the approach to Inverse Optimal Control embraced in this chapter establishes a connection to recent development in *Generative Adversarial Networks*, optimal transport, and the general approach of *moment matching*.

7.1 Decisions are Purposeful: Inverse Optimal Control

7.2 IOC as Moment Matching

Let us consider the problem of matching such expert driver behavior. What would success mean here? A reasonable requirement might be that:

$$p(\xi|\Gamma) \approx \tilde{p}(\xi|\Gamma)$$

where we use ξ to represent a trajectory the expert might take, Γ to represent the general context of the planning problem including environment and sensor data at a particular time-step, we use p to represent our model distribution and we use $\tilde{p}$ to represent the empirical distribution of examples.

A weaker condition that we'll work with here is that,

$$E_{p(\tilde{\Gamma})}[p(\xi|\Gamma)] \approx E_{p(\tilde{\Gamma})}[\tilde{p}(\xi|\Gamma)]$$

Throughout the remainder of this work, we'll largely suppress the dependence on the context Γ. Near the end, we'll consider stronger notions than the average case performance.

I'll ignore here the philosophical quandaries around the demonstrations coming from a probability distribution. They don't. It's simpler to imagine they do for the purposes of this work and defer a generalization to regret and other non-probabilistic notions of performance to another time.

How, then, should we make it so that $p(\xi) \approx q(\xi)$ (where in general q is a distribution, often the empirical one $\tilde{p}$) in a manner that is empirically measurable, and where we only have sample access to q (and might prefer for engineering reasons to only require sample access to p)? Let us consider first a finite set of *cost functions* $\mathcal{F}$ that measure, according to some notion, how good a trajectory is. We might then require that the demonstrations and the learned model achieve the same costs for each of the cost functions: [1]

$$\forall f \in \mathcal{F}, E_p[f] = E_q[f] \tag{7.2.1}$$

[1] We might prefer, instead that the learned model has a **lower** cost, $E_{p(\tilde{\Gamma})}[p(\xi|\Gamma)] \leq E_{p(\tilde{\Gamma})}[\tilde{p}(\xi|\Gamma)]$. This is also easy to implement, (and covered in generality in) but if $\mathcal{F}$ is closed under negation ($f \in \mathcal{F} \Rightarrow -f \in \mathcal{F}$) then meeting the inequality immediately implies we meet the equality.

K. Waugh, B. D. Ziebart, and J. A. Bagnell. Computational rationalization: The inverse equilibrium problem. In *Proceedings of the International Conference on Machine Learning*, June 2011

Moment matching

This constraint, Equation 7.2.1, is known as a moment matching constraint. We note the important property that moment matching implies that for the much broader class of cost functions composed of linear combinations of $\mathcal{F}$, $\mathcal{F}_{\text{lin}} = \{\sum_i \lambda_i f_i | f \in \mathcal{F}\}$, we also have

$$\forall f \in \mathcal{F}_{\text{lin}}, E_p[f] = E_q[f] \tag{7.2.2}$$

That is, moment matching under a set of cost functions, also implies matching all linear combination of such moments.

Prove it in the margin.

Classes of moments. The classical moments are the monomials, e.g. $x_1^2 x_3^4 x_5$. Classic results, perhaps unsurprisingly, indicate matching all monomial moments implies convergence in distribution. Matching all bounded functions implies that the total variation distance between distributions is 0. [2]. A more classic set of functions relevant to robot motion planning including quadratic hinges on an individual variable [3] $\max^2(0, x_i)$, or generalized to $\max^2(0, g(x, \Gamma))$ for a class of functions g.

[2] Bharath K. Sriperumbudur, Arthur Gretton, Kenji Fukumizu, Gert R. G. Lanckriet, and Bernhard Schölkopf. A note on integral probability metrics and ϕ-divergences. 2009. URL http://arxiv.org/abs/0901.2698

[3] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa. Chomp: Gradient optimization techniques for efficient motion planning. In *ICRA*, 2009b

From moments to metric

We can allow slack in the notion of moment matching and provide a distance metric (divergence) between probability distributions. We denote by

$$D_{\mathcal{F}}(p, q) = \max_{f \in F} E_p[f] - E_q[f] \tag{7.2.3}$$

the *Moment Matching Metric* also known as the *Integral Probability Metric* associated with $\mathcal{F}$. That is, we measure the difference between distributions by considering the worst-case gap between them in the class of cost functions $\mathcal{F}$.

We consider here the case that the class $\mathcal{F}$ is symmetric and I blithely ignore the difference between a supremum and a maximum.

If the function class is a Reproducing Kernel Hilbert Space of fixed norm, this metric is known as Maximum Mean Discrepancy [4], while if the class of functions is all 1-Lipshitz continuous ones, this metric is equivalent to the *earth mover distance*.

[4] Bharath K. Sriperumbudur, Arthur Gretton, Kenji Fukumizu, Gert R. G. Lanckriet, and Bernhard Schölkopf. A note on integral probability metrics and ϕ-divergences. 2009. URL http://arxiv.org/abs/0901.2698

Measuring by samples

This definition of divergence between distributions is useful both because it captures the important distinctions in a concrete notion of cost, but also because expectations can be measured by only observing samples. If we have, for instance, a single f to consider in our cost function space, the strong law implies immediately $\frac{1}{N}\sum_i f(\xi_i) \to E_p[f]$ for paths drawn from $p(\xi)$,

and we can bound the error in this estimate with high probability, typically as a rate of $\epsilon = O(\frac{1}{\sqrt{N}})$. Intuitively this follows from the observation that the variance of a sum of *i.i.d.* random variables is linear in the number of samples and hence that the standard deviation of an average of random variables must decrease as $O(\frac{1}{\sqrt{N}})$.

For broad classes of cost functions we are basically solving a classical supervised learning problem: finding the function f that maximally distinguishes between p and q. If we take a gradient of equation 7.2.3 with respect to the function f and then take a random instance, we find an (unbiased) estimate of the gradient is:

$$f(\xi_p)k(\xi_p, \cdot) - f(\xi_q)k(\xi_q, \cdot)$$

in an RKHS— replace k with a delta function or an indicator function as appropriate for the space), where f is the element of $\mathcal{F}$ that is the current linearization point.

Entropy Regularization

In general, we can only approximately estimate moments from samples. For a finite number of moments, there will be many distributions that are consistent with the moments known. A classic approach to breaking this ambiguity is the maximum entropy method that prescribes assigning the highest entropy distribution consistent with the known moment constraints. That are many justifications of this principle (see [5]) that are more-or-less compelling depending on applications. Such regularization has proven critical in inverse optimal control applications. The probabilistic viewpoint leads to a well-defined answer to modeling imperfect "optimal" behavior while managing the ill-posedness of equally good solutions.

The result sets up an optimization problem:

$$\max_p H[p(\xi)] \tag{7.2.4}$$

$$s.t. \forall f \in \mathcal{F}, \tag{7.2.5}$$

$$E_p[f(\xi)] = E_q[f(\xi)] \tag{7.2.6}$$

where q is typically the empirical distribution over observed paths, $\tilde{p}$.

Is it easy to relax the equality above with slack via the moment matching metric,

$$\max_{f \in F} E_p[f] - E_q[f] \leq \epsilon$$

We defer that now, except to note that slack in the primal, MaxEnt problem, corresponds to regularization in the dual parameters – a beautiful result due to Dudik et al. [2004].

The resulting Lagrangian optimization problem is a game between two players. A generator $p(\xi)$ that computes the best distribution, and a cost function (weighting) λ attempts to discriminate between the two distributions. Some techniques, like GANs, attempt to solve the problem by a saddle point finding approach. This is a potentially powerful approach for IOC [6]. However, the industry standard, if you'll oblige, is to solve for the optimal generator p in closed form. Given a function class F_{lin} that is closed under linearity, we can conclude

$$p(\xi) = \frac{exp(-\text{cost}(\xi))}{Z} \tag{7.2.7}$$

[5] E. T. Jaynes. *Probability theory: The logic of science*. Cambridge University Press, 2003; and Peter D. Grunwald and A. Philip Dawid. Game theory, maximum entropy, minimum discrepancy and robust bayesian decision theory, 2004. URL http://arxiv.org/abs/math/0410076 Interestingly, such entropic regularization occurs in purely computational settings as well from AdaBoost to Sinkhorn iterations where it break ambiguities and leads to fast, numerically stable algorithms.

It's interesting to note that the duality viewpoint suggests that we should often actually measure a divergence not by its maximum over $\mathcal{F}$, but by the L_2 norm of the violations. I'm unaware of that being explored in the probabilistic literature, but it is much more natural notion of approximate moment matching than IPMs for many applications.

[6] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. 2016. URL http://arxiv.org/abs/1606.03476

where in the linear case for a set of features of a path $\phi(\xi)$, we have cost $= \lambda^T \phi(\xi)$.

Given the form, the goal now is only to compute the cost function, or equivalently it's parameters (λ).

Derive this in the margin using Lagrange multipliers.

Markov structure

We being by considering a problem with structure defined by cost $f(\xi) = \sum_t cost(x_t, u_t)$ and markov transition dynamics $x_{t+1} = h(x_t, u_t)$. For the linear case we write such costs as $\lambda^T \phi(x, u)$, for a set of feature functions ϕ. This structure makes it (exponentially in T) more efficient to find solutions and corresponds to the classical structure of optimal control.

We note in all cases there is nothing essentially different about making each weight or cost a function of t and that this could be useful for receding horizon control where our uncertainty about the future grows rapidly. [7]

If we want to compute derivatives with respect to λ, we should find the optimal generator p and then eliminate to form a dual optimization depending only on the costs (i.e. only on the variable λ rather than p). A quick computation of the derivatives with respect to λ after eliminating p gives us:

$$\sum_t E_p[\phi(x_t, u_t)] - E_q[\phi(x_t, u_t)]$$

where $E_q[\phi(x_t, u_t)]$ is typically a constant estimated by observed data (i.e. $q = \tilde{p}$). Optimization then boils down to computing expectations under the model efficiently $E_p[\phi(x_t, u_t)]$. We can do this via a dynamic programming algorithm (effectively inference in a random field) which is precisely equivalent to classical value iteration with the min replaced by $\log \sum \exp$, aka *softmin*. The forward pass can then be computed analytically or via samples by noting

$$p(u_t|x_t) \propto \exp(-Q_t(x_t, u_t))$$

where $Q_t(x_t, u_t) = \text{cost}(x_t, u_t) + V_{t+1}(x_{t+1})$ and $V_{t+1}(x_{t+1}) = \text{softmin}_u Q_{t+1}(s_{t+1}, u)$. This can then be pushed through forward dynamics. In general, we can view the backwards pass as transforming an undirected graphical model into a directed one, and then employing an ancestral sampling procedure (or exact forward integration) to compute expectations.

This recursive definition enables an optimal policy computation for a tabular model and suggests methods to enable approximation in the non-tabular case.

Scaling to continuous trajectory generation

We're typically interested in high dimensional continuous control problems. Some powerful tools exist here for inference including Monte-Carlo methods and Vernaza's value-function symmetry method[8]. The simplest version however is the Linear-Quadratic / Gaussian model. This was first developed by [9] (following Ratliff's thesis version of IOC for LQR). We can follow the development presented in Ziebart, or take an alternate approach of approximation. In particular, we can consider the Laplace approximation where we find the most probable trajectory ξ^* and a quadratic expansion about it to compute approximate partition functions, entropy, variances, samples and

The general case of stochastic dynamics is handled in Ziebart et al. [2013], and requires more careful reasoning about causal entropy, but the resulting algorithms are largely identical.

[7] Having variable costs as a function of time (at least without exponential damping) can, however, lead to inconsistent decision making. For instance, an agent might always choose to defer expensive decisions to the future, where costs are lower, as during training, "the future never arrives". This is closely related to the problems of off-policy imitation described in the previous chapter.

[8] P. Vernaza and D. D. Lee. Efficient dynamic programming for high-dimensional, optimal motion planning by spectral learning of approximate value function symmetries. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2011

[9] B. D. Ziebart, J. Andrew Bagnell, and A. K. Dey. Modeling interaction via the principle of maximum causal entropy. In *Proceedings of the 27th International Conference on Machine Learning*, 2010

expectations. Under linear dynamics and a quadratic cost function, the true distribution on both actions $p(u_t|x_t)$ and on states are Gaussian and thus the approximation is actually exact.

In the absence of these, the technique is an approximation, albeit a powerful one. The key idea is to leverage a family of techniques based on *Differential Dynamic Programming* [10] to compute efficiently the best (and thus most probable) action as a policy $u_t|x_t$ and the curvature in the action-value function as a function of u, denoted Q_{uu}. This quadratic approximation in value (cost-to-go) provides a Gaussian approximation of action-selection which enables both efficient inference and easing understanding.

Note crucially that such methods as DDP (and iLQR and variants) provide a complete feedback policy and an estimate of the cost-to-go that depends on the state we arrive in. Equivalently, in probabilistic terms, they provide a sequence of ancestral conditional distributions rather than merely marginals. [11]

We begin below by describing a Laplace approximation based sampler. It assumes a differential dynamic programming procedure has already been called that provides gain matrixes and control biases, as well as curvatures. The notation here matches that used by Tassa et al. [2012].

```
# Take an initial state x_0, a model of forward dynamics f: state, action -> state,
# a sequence [] of Quu curvature approximations (conditional partition functions), and
# gain matrices/vectors K,k computed via a Differential Dynamic Programming
# method, and a maximum time T
def Sample(x_0, f, Quu, K, k, T):
    x = [x_0] # state sequence
    u = [] # action sequence
    for t in range(T):
        π = Normal(k[t] + K[t] x[t], Quu(t)^-1)
        u.append(Sample(π))
        x.append(f(x[t],u[t]))
    return (x,u)
```

LQR forward sampler

We note however, that in the spirit of receding horizon, model-predictive control, we can actually do better. Once we have begun sampling, we can re-linearize and resolve the optimal control problem that results from arriving at state x_1. This suggests an $O(T^2)$ procedure which samples ancestrally and recomputes the optimal gain matrices and curvatures as it advances. We outline this in pseudo-code below.

```
# Take an initial state x_0, a model of forward dynamics f: state, action -> state,
# a cost function cost: state, action -> R,  a maximum time T,
# and a procedure DDPSolve: f, cost, state, integer --> a tuple of
# ([gain Matrix K],[gain vector k],[quadratic approximation of Q_uu] ) , each a time varying
# sequence computed via a Differential Dynamic Programming method.
# It returns a single sample trajectory of states and controls.
def Sample(x_0, f, cost, T, DDPSolve):
    x = [x_0] # state sequence
    u = [] # action sequence
    for t in range(T):
        (k,K,Q)= DDPSolve( f, cost, x[-1], T) # compute optimal gain matrices
        π = Normal(k[0] + K[0] x[0], Quu[0]^-1)
        u.append(Sample(π))
        x.append(f(x[0],u[0]))
    return (x,u)
```

Re-linearized MPC LQR forward sampler

In the above procedure, care must be taken with passing around a context Γ: the context should remain unchanged during the sampling and all time-varying costs must be indexed according to the correct time, as the *DDPSolve()* routine is unaware that the time-steps are changing in the outer loop of the algorithm.

[10] D. H. Jacobson and D. Q. Mayne. *Differential Dynamic Programming*. Elsevier, 1970; C. G. Atkeson. Using local trajectory optimizers to speed up global optimization in dynamic programming. In *Advances in Neural Information Processing Systems (NIPS)*, 1994; and Yuval Tassa, Tom Erez, and Emanuel Todorov. Synthesis and stabilization of complex behaviors through online trajectory optimization. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 4906–4913. IEEE, 2012

[11] This point turns out to be an important difficulty in creating constraints in LQR or in probabilistic inference algorithms— in either case many methods lose they key advantage of a feedback policy and regress to only optimization. LQR is not only optimization– it's a full policy. This point is addressed further below.

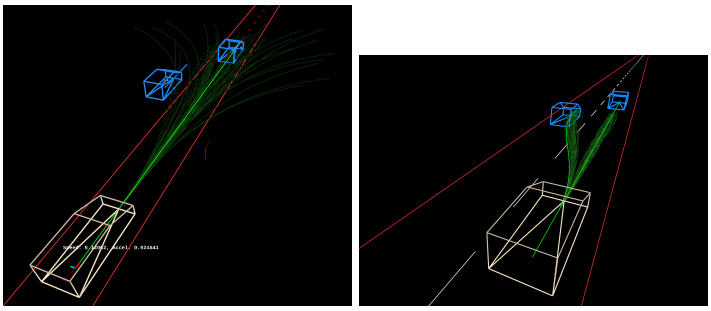


Figure 7.2.1: Both images depict sample trajectories drawn from a Gaussian approximation, computed via Differential Dynamic Programming, of the maximum entropy distribution for a an autonomous vehicle. The left image shows the naive Gaussian sampling procedure. Note the expansion in time at approximately $O(\sqrt{T})$ of the lateral position of the vehicle. The red boundaries shown, which are expensive to violate, do not impact the mode trajectory, and therefore they do not affect the Gaussian sampling approximation. In the right figure, we explore the use of a sampler that iteratively re-linearizes and resolves after each step in ancestral sampling from x_0. By contrast, we see two basins (corresponding to lanes), and we note the lateral position of the trajectories is strongly influenced by boundaries.

Learning cost functions from samples

We have already seen how a straightforward gradient can be computed from the difference of $E_p[\phi]$ and its empirical version $E_{\bar{p}}[\phi]$. Let's consider expanding to a space of functions and learning using samples. We begin by replacing $\lambda^T \phi(x, u)$ by $\text{cost}(x, u)$ from a linear space of cost functions (that is, closed under linear combination).

There are three general strategies for learning such cost functions, and perhaps surprisingly, they are all actually closely linked. The first two can be understood as generically gradient descent in a space of functions (1) Boosting and (2) Kernel Gradient Descent, while the final one (3) parametric gradient descent in a function class, is the older and at times most computationally efficient approach. [12]

Taking the MaxEnt family $p(\xi) \propto \exp -cost(\xi)$ and plugging it into the maximum entropy problem Equation 7.2.4 yields a maximum likelihood problem over the space of functions. [13] An unbiased estimate of that gradient can be computed in approach (3) as

$$\nabla_\theta \text{cost}(\xi_i) - \nabla_\theta \text{cost}(\xi_{\text{sample}}) \tag{7.2.8}$$

where ξ_i is a demonstration sample and ξ_{sample} is drawn from the model $p(\xi)$. Note the cost function being additive over time, this turns into a batch of updates, one for each time-step. The functional versions of this procedure simply generate datapoints indexed by x, u and a positive or negative regression target. This is demonstrated below for a particular variant of boosting.

The result is at heart a two-sample test and update procedure for learning cost functions that is essentially equivalent to the Learning to Search procedure outlined. We use here samples of model behavior rather than simply the most probable/optimal trajectory as in the original algorithm description. The approach is also closely connected to *Generative Adversarial Models* (GANs). The key difference is that GANs, unlike the exponential family or IOC methods, don't solve for the optimal generator in closed form, but instead update an approximate generator. As a result, inference requires only forward execution of the generator model and may be better behaved for singular/constrained-to-a-manifold distributions. The cost is that inference doesn't enable building in engineered constraints or cost-function insight.

Solving the game A critical note is that we are indeed solving a game between generator/policy and discriminator/cost function so we must carefully control step size in learning a cost function. Thus it is particularly

Kernel gradient descent is nearly the same as boosting with a particular class of weak learners, while in a particularly important, infinite-width limit, parametric gradient descent on a deep, non-convex function class behaves precisely as a kernel gradient descent including convergence to a global optimum and appropriate regularization guarantees.

Arthur Jacot-Guillarmod, Franck Gabriel, and Clement Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 8580–8589. Curran Associates, Inc., 2018. URL http://papers.nips.cc/paper/8076-neural-tangent-kernel-convergence-and-general.pdf

[13] Check it!

important in the family of optimization algorithms above to take constrained steps, or regularize to prior solutions. The optimization objective for a fixed inference policy is linear, and hence will run away to infinity in the cost function update. This is the classic problem of a learning in a game– we have a stable strategy applied for the outer player while the inner player (policy generation) is best response. Boosting style methods do this automatically via their additive model and step size control.

Structured Cost Families

It's quite natural to write down parametric families of cost functions like $\phi(x,u) = \max^2(0, x_i - g_\theta(x_0, \Gamma))$ (for a context variable Γ) where we are attempting to constrain the form of the objective function to be nicely behaved (single global optimum, strong curvature, multiple derivatives everywhere) in a variable of interested. This encourages inference to remain efficient and enables building in engineering insight in cost-function design.

7.3 LEARCH generalized.

In the previous chapter, we discussed a *functional gradient* approach to solving the Structured Maximum Margin formulation of Inverse Optimal Control. That general non-parametric learning strategy is equally applicable to the MaxEnt framework, as is a parametric "deep variant" enabled through backpropagation.

We may step through the LEARCH-MaxEnt algorithm on a cartoon example to see why it might work. We first consider a path driven by teacher from a start point to a goal point, then imagine a simple planning problem on a discretized grid of states that the robot can occupy. Every iteration of the algorithm consists of the following: (a) computing a sample plan/policy from our approximate MaxEnt distribution; (b) identifying where the plan and teacher disagree and creating a data set consisting of features and the direction in which we should modify the costs; (c) using a supervised learning algorithm to turn that data set into a simple predictor of the direction to modify costs; and (d) recomputing a cost function as a (weighted) sum of the learned predictors.

```
# Take a sequence of trajectory optimization problems and demonstrations [(M_i, ξ_i)]_{i=1}^N where MDP M is a
      planning problem consisting of states, actions, and a transition function used for planning,
# feature function f: state, action -> R^d that describes states in terms of features meaningful for
      cost
# α, a step-size (which can be generalized to a shrinking sequence)
# Relies on procedures to initialize the cost function,
# and to build an optimal maxent policy, and
# to sample that policy

def MaxEntLEARCH({(M_i, ξ_i)}_{i=1}^N, f, α):
  s_0 = init() #Initialize cost function, s_0 : R^d → R
  for t in range(T): # run for T iterations
       D = [] # Initialize the training data set to empty
       for i in range(N): # for each example in the data set
            c_i = s_t(f) # Compute cost function for this problem
            π_i^* = Optimize(M_i, c_i) # find (approximately) the resulting MaxEnt policy π_i^*(x)
            #      that leads to p(ξ) ∝ exp Σ_{(x,u)∈ξ} c_i(x,u)
            μ^* = Sample(M_i, π_i^*)
            # μ^*'s contains the state action pairs from a random trajectory created by the Sampler.
            # More sophisticated samplers might interleave sampling and "optimization".
            # Generate positive and negative training examples:
       D_i^pos = [ (f_i(s,a), 1) for (s,a) in μ^* ]
       D_i^neg = [ (f_i(s,a), -1) for (s,a) in ξ_i ]
       # if the same state occurs in both samples, we can remove it
       D.append(D_i^pos)
       D.append(D_i^neg)
```

```
        h_t = Learn(D) # Train a regressor (or classifier) h_t: R^d->R on the resulting data set
        # The Data Aggregation is not required but does at times lead to more stable performance
        s_{t+1} = s_t + α_t h_t # Update the hypothesis cost function
    return s_T
```

MaxEnt LEARCH Algorithm Pseudo-code

As with the previous LEARCH algorithm, we initialize the algorithm by guessing at a cost function: for instance, by assuming a constant cost everywhere. Instead of a planner, we run the sampler that generates trajectories. We can identify where the sample path agrees and disagrees with a demonstration by a teacher of the correct path. Again, we create a data-point that contains the features that describe the state and assign it a target value to increase or lower the cost depending on whether the sample or the teacher's path traverses that location.

The same procedure is run for locations of disagreement across multiple trajectories (that is multiple planning problems). The resulting data set is then handed to a supervised learning algorithm (linear regression, Support Vector machines, a neural network) that produces a new predictor which maps features to a scalar cost.

As with any boosting style algorithm, the proposed cost function is simply the old cost function added to the new predictor, and we continue to update it by adding in new components.

MaxEnt Relation to Maximum Margin Planning

If we consider the limiting case of $\text{cost}(\xi)_{\text{temp}} = \frac{\text{cost}(\xi)}{T}$ as $T \to 0$, and use the gradient/boosting rules above we recover the max-margin approach to cost function generation. This approximation is less robust (although can prove very useful!) as it tends to lead to cost function collapses. Intuitively, this occurs as we see demonstrations that are highly sub-optimal and we can only generate optimal samples. These will tend to be lower in every element of the feature vector $\phi(x, u)$ and hence the gradient will continue to shrink the cost. No cost shrinkage, however, leads to higher entropy behavior, and thus the costs can collapse. As such, for sub-optimal demonstrations, Maximum Entropy should be preferred whenever it is appropriate. The LQR/Laplace approximation dramatically increases the places where that is possible, as sampling from the model is no more expensive than optimization.

7.4 Anomaly Detection

We can identify important actions (or trajectories) by computing

$$\log p(\xi_{\text{example}}) = -\text{cost}(\xi_{\text{example}}) - \log \sum_{\xi} \exp\left(-\text{cost}(\xi)\right)$$

where the second term comes from the normalizing constant Z and is the softmin of the path costs. A rough and ready approximation is to simply compute

$$\log p(\xi_{\text{example}}) \approx -\text{cost}(\xi_{\text{example}}) + \min_{\xi} \text{cost}(\xi)$$

which requires only access to an optimal planner rather than sampling or computing the complete partition function. Demonstrations that have highly negative log-probabilities should be considered as outliers— the model poorly captures the behavior.

Refined estimates via Laplace Approximation We can, of course, also use the Laplace (Gaussian) approximation to estimate how many standard deviations a particular control variable is from from the expected (mode/mean) control. This also makes it easy to make more refined analysis of outliers and detect plans that are, say, k-standard deviations away from the mean in one control or state variable axis. For instance, if we have a robotic system with a clear longitudinal control mode, we can identify points that are 3σ away in negative longitudinal control. Such anomalies (hard deceleration) can be important to identify.

We can also plot (or statistically test) the Gaussian approximation of any state variable and test how far wrong our current model is on any individual state variable.

Gradients estimates. Given we know that $\phi(x_{\text{example}}, u_{\text{example}}) - \phi(x_{\text{sample}}, u_{\text{sample}})$ is an unbiased estimate of the gradient, we know it should on average be 0 if our model is well fit. Plots of the elements of these gradients provide clear signal of model under-fitting or failure to set costs appropriately.

Forecasting. Work by [Baker et al., 2009] demonstrates people reason about others as deliberative agents as well. This inverse planning framework elegantly captures aspects of the human "Theory of Mind." Work in operations research and econometrics, particularly by Rust [Rust, 1992, 1994], derives predictive distributions by developing a framework for learning cost functions and predictive stochastic policies for agents acting according to a Markov Decision Process (MDP). Intriguingly, the same policy structure and dynamic programming algorithms derived from a maximum entropy formulation are developed from considering an economist with only partial access to the prediction problem and including "shocks" in a model of what would otherwise be optimal behavior. These close connections between operations research, control theory and machine learning deserve much deeper investigation.

7.5 Test-time Costs and Constraints

An important power of the approach of model-based IOC, as opposed to naive policy learning techniques, is the freedom to add new constraints and cost functions not present in the training data or that are critical to enforce at test time. Said differently, when generating trajectories for use in anger, we may wish to further shape these beyond what is represented directly in the data.

The planning-based approach allows a powerful combination of engineering design and machine learning integrated through test-time optimization. The price we pay, of course, is in needing to solve a potentially complex optimization problem at test time, and the unfortunate fact that efficient 2nd order dynamic programming solutions aren't available in industry-standard *differentiable programming frameworks* like Tensorflow or Torch.

We also note that it is often important that test time inference actually be the most probable trajectory, or at least the temperature lowered dramatically in sampling. This is imperative for a few reasons, notably that: a) we usually want the best solution at test time, not a sample one and b) likelihood and

entropy based models are strongly incentivized to "cover" and explain the data available to them rather than simply provide the optimal generation. [14]

[14] Massimo Caccia, Lucas Caccia, William Fedus, Hugo Larochelle, Joelle Pineau, and Laurent Charlin. Language gans falling short. *CoRR*, abs/1811.02549, 2018

Policies, Probabilities, and Constraints

An important part of both the Bellman Equation (in its many instantiations including LQR) and the MaxEnt Inverse Optimal Control formalism is that it computes policies rather than trajectories. A notable difficulty arises in combining either technique with hard constraints. We can use primal-dual (lagrangian) methods for identifying the most probable trajectory, but both feedback policies and conditional distributions break down with these techniques. It's unclear if it's possible to achieve the best combination of primal-dual and policy/probabilistic approaches. This failure tends to privilege reparameterization based techniques as both feedback and uncertainty can still be sensible. These tend to require more careful design than primal-dual methods, at least for strictly enforcing constraints.

Acknowledgements

We thank colleagues Brian Ziebart, Arun Venkatraman, and Wen Sun for tremendous insight in this area and many productive conversations. The images of sampling and the implementation used for differential dynamic programming are due to Arun Venkatraman.

8
Approximate Dynamic Programming

Approximate Dynamic Programming (ADP), also sometimes referred to as *neuro-dynamic programming*, attempts to overcome the limitations of value and policy iteration in large state spaces where some *generalization* between states and actions is required due to computational and sample complexity limits. Further, all the algorithms we have discussed thus far require a strong *access model* to reconstruct the optimal policy from the value function and to compute the optimal value function at all. We'll consider algorithms in the rest of these lecture notes that relax this strong notion of access.

Access Models

Up until this chapter, we've implicitly largely assumed that we have complete, white box access to a full description of the system dynamics for the purposes of applying dynamic programming. In practice, reinforcement learning problems differ by the degree of "system access" that is available. For the Tetris problem we often assign as homework for this class, we can recreate the exact same state over and over again while learning (or testing our algorithms). For robotic systems, we typically have a weaker form of access – we can never create *exactly* the same state again, but we can often run multiple trials. It's worth reviewing here some notions of access model for a system as the techniques we can apply and which will be most effective are largely governed by this access. We review them in order of the strength of the model access; each of the earlier access models can trivially simulate the ones below it, but not (necessarily) visa-versa.

1. Full Probabilistic Description

   In this model, we have access to an explicit cost function and the transition function for every state-action written down as a large matrix that can be manipulated. A major downside of having this kind of model is that it easily can become so large as to be computationally intractable for a non-trivial problem. It is also information-theoretically hard to identify this type of model from data– it simply isn't possible to visit a very large state-space.

2. Deterministic Simulative Model In the simplest version, we have a function that maps $f(x, a) \rightarrow x'$, deterministically. [1] More broadly, a deterministic simulative model can mean that we while the dynamics are stochastic, we have access to the random seed in a computer program, so

[1] We'll assume through the discussion below that the cost function has the same access. It can be the case, however, that we can be "in between" such models. For instance, we might have a more complete description of a cost function as a quadratic, while only having *reset model* access to the dynamics.

we can recreate trajectories including the randomness that occurred. Such access is what is typically available in computer simulations. [2]

3. Generative Model
   In this model, we have programmatic access. We can put the system into any state we want.

4. Reset Model
   In this model, we can execute a policy or roll-out dynamics any time we want, and we can always reset to some known state or distribution over states. This is a good model for a robot in the lab that can be reset to stable configurations.

5. Trace Model
   This is the model that best describes the real world. Samuel Butler said "Life is like playing a violin solo in public and learning the instrument as one goes on"; the trace model captures the inability to "reset" in the real world.

[2] Although, unfortunately, non-determinism in simulators is more prevalent than one might expect.

There are a few general strategies one can pursue for applying approximation techniques.

Approximate the Algorithm. The most straightforward approach is to take the algorithms we've developed thus far *Policy Iteration* and *Value Iteration*, and replace the steps where we would update a tabular representation of the value function with a set of sampled (*state-action-next state*) and a supervised-learning *function approximator*.

This approach is an incredibly tempting way to pursue hard RL problems: we simply plug in a regression estimator and run existing, known-to-be-convergent algorithms. In a sense, we can see the tremendously successful Differential Dynamic Programming approach as of this form: we are finding quadratic approximations and running the existing value-iteration approach.

We find below that while at times successful in practice, there are many sources of *instability* in these algorithms that result in often extremely poor performance. We analyze informally the two main sources of error: the *bootstrapping* that happens in dynamic programming mixes poorly with generalization across states, and even more significantly, the change of policy induced by the max operation produces a *change in distribution* (affects which state-actions matter most) that dramatically amplifies any errors in the function approximation process. We discuss some strategies for remediating these.

Approximate the Bellman Equation. The next broad set of strategies is to treat the Bellman equation itself as a fixed point equation and optimize to find a fixed point. These techniques, known as *Bellman Residual Techniques* are dramatically more stable and have a richer theory.[3] [2] Practically, the performance is often (but not always!) worse than methods based on the "approximate the dynamic programming" strategy above, and it suffers as well from the *change of distribution* problem.

Approximate the Policy Alone. We cover a final approach that eschews the bootstrapping inherent in dynamic programming and instead caches policies and evaluates with rollouts. This is the approach broadly taken by methods like *Policy Search by Dynamic Programming* [4] and *Conservative Policy Iteration*[5]. [6]

[3] L. C. Baird. Residual algorithms: Reinforcement learning with function approximation. In *International Conference on Machine Learning*, 1995; and Wen Sun, Geoffrey J Gordon, Byron Boots, and J Bagnell. Dual policy iteration. In *Advances in Neural Information Processing Systems*, 2018

[4] J. A. Bagnell and J. Schneider. Covariant policy search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2003

[5] S. Kakade and J. Langford. Approximately optimal approximate reinforcement learning. In *Proceedings of the 19th International Conference on Machine Learning (ICML)*, 2002

[6] Methods like the Natural Policy gradient approach that we discuss later are closely connected.

Action-Value Functions

In this lecture, we consider the finite horizon case with horizon T. The *quality function*, Q-function, or *action-value function* is defined as,

$$Q^*(x,a,t) = c(x,a) + \text{total value received if optimal thereafter,}$$
$$Q^\pi(x,a,t) = c(x,a) + \text{total value received if following policy } \pi \text{ thereafter.}$$

These can be restated in terms of the Q-function itself as

$$Q^*(x,a,t) = c(x,a) + \gamma \mathbb{E}_{p(x'|x,a)}[\min_{a'} Q^*(x',a',t+1)]$$
$$Q^\pi(x,a,t) = c(x,a) + \gamma \mathbb{E}_{p(x'|x,a)}[Q^\pi(x',\pi(x'),t+1)]$$

Note that unlike infinite horizon case where a single value function/action-value function is defined, there are T value functions/action value functions for the finite horizon case, one for each time step.

Once we have the action-value functions, the value function V^* and the optimal policy π^* are easily computed as

$$V^*(x,t) = \min_{a \in \mathbb{A}} Q^*(x,a,t)$$
$$\pi^*(x,t) = \operatorname*{argmin}_{a \in \mathbb{A}} Q^*(x,a,t)$$

We can compare the above equation to how we compute the optimal policy from the optimal value function,

$$\pi^*(x,t) = \operatorname*{argmin}_{a \in \mathbb{A}} c(x,a) + \gamma \mathbb{E}_{p(x'|x,a)}[V^*(x',t+1)]$$

Pros and Cons of Action-Value Functions

Pros

1. Computing the optimal policy from Q^* is easier compared to extracting the optimal policy from V^* since it only involves an argmax and does not require evaluating the expectation and thus the transition model.
2. Given Q^*, we do not need a transition model to compute the optimal policy.

Cons

1. Action-value functions take up more memory compared to value functions (|States| x |Actions|, as opposed to |States|).[7]

[7] Note, however, that if we use a value function instead of Q-function, we may need another |States| x |Actions| table to store the transition probability in order to find the optimal policy if the transition model is not given analytically.

Fitted Q-Iteration

We can now describe Fitted Q-Iteration, an approximate dynamic programming algorithm that learns approximate action-value functions from a batch

of samples. Once the data is collected the Q-function is approximated without any interaction with the system.

Algorithm 10: Fitted Q-iteration with finite horizon.

Algorithm `FittedQIteration`$(\{x_i, a_i, c_i, x_i'\}_{i=1}^n, T)$
  $Q(x, a, T) \leftarrow 0,\ \forall x \in \mathbb{X}, a \in \mathbb{A}$
  forall $t \in [T-1, T-2, \ldots, 0]$ **do**
    $D^+ \leftarrow \emptyset$
    forall $i \in 1, \ldots, n$ **do**
      input $\leftarrow \{x_i, a_i\}$
      target $\leftarrow c_i + \gamma \min_{a'} Q(x_i', a', t+1)$
      $D^+ \leftarrow D^+ \cup \{\text{input}, \text{target}\}$
    end
    $Q(\cdot, \cdot, t) \leftarrow \textsc{Learn}(D^+)$
  end
  return $Q(\cdot, \cdot, 0 : T-1)$

The algorithm takes as input a data-set D which contains samples of the form {state, action, associated cost, next state}. In practice, this is obtained by augmenting expert demonstration data with random exploration samples. As in value iteration, the algorithm updates the Q functions by iterating backwards from the horizon $T-1$. Essentially, for each time step t, we create a training data-set D^+ by using the learned Q function learned for time step $t+1$. This data-set is fed into a function approximator LEARN, which could be any of your favorite machine learning models (linear regression, neural nets, Gaussian processes, etc), to approximate the Q function from the training dataset. We could also start with an initial guess for $Q(\cdot, T)$. [8]

[8] The version presented here assumes the dynamics and cost functions are the same at each time-step.

Note that the above fitted Q-iteration algorithm can be easily modified to work for infinite horizon case. In fact, the infinite horizon version is simpler, because we can choose to maintain a single Q function. Hence, for each iteration, we can just collect a batch of samples, and update the Q function.

Algorithm 11: Fitted Q-iteration with infinite horizon.

Algorithm `FittedQIteration`$(\{x_i, a_i, c_i, x_i'\}_{i=1}^n)$
  $Q(x, a) \leftarrow 0,\ \forall x \in \mathbb{X}, a \in \mathbb{A}$
  while *not converged* **do**
    $D^+ \leftarrow \emptyset$
    forall $i \in 1, \ldots, n$ **do**
      input $\leftarrow \{x_i, a_i\}$
      target $\leftarrow c_i + \gamma \min_{a'} Q(x_i', a')$
      $D^+ \leftarrow D^+ \cup \{\text{input}, \text{target}\}$
    end
    $Q \leftarrow \textsc{Update}(Q, D^+)$
  end
  return Q

There are a few of things that we need to be aware of when using fitted Q-iteration in practice:

- In a goal-directed problem, we need to make sure that our samples include goal states in order to get meaningful iterations.
- Often it makes sense to run the algorithm on features of the state-action

pair (x, a), not the raw state-action pairs themselves.

- Fitted Q-iteration can be run repeatedly, augmenting the data set with new samples from the resulting policies.
- For goal-directed problems, the goal states x_i are nailed down to 0 Q-value (target = c_i), and bad or infeasible states are provided a large constant target value c^-. The former ensures that the Q-values do not drift up over time, and the latter prevents the Q-value for bad states from blowing up to ∞.
- Value functions are not smooth in general (e.g. mountain car problem). A simple trick to fix this is to add noise to the transition model, which smooths out discontinuities.

Case Study

A robotics example of work using Fitted Q-Iteration is demonstrated in [9]

[9] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.3532&rep=rep1&type=pdf

The authors demonstrate that a neural (in modern parlance, "deep") fitted Q-learning algorithm can learn a control strategy from scratch for driving a car along a GPS-guided course, minimizing cross-track-error (distance of vehicle to one side of a straight line between waypoints). All data for learning came from actual driving; i.e. there is neither a model nor a use of data-augmentation. As has been, perhaps surprisingly, common in robotics, the actual network is a relatively shallow 3 layer neural net for regression. This contrasts with work for imitation learning of driving controllers like that of [10] where very deep networks were used.

[10] M. Nechyba and J. A. Bagnell. Stabilizing human control strategies through reinforcement learning. In *Proc. IEEE Hong Kong Symp. on Robotics and Control*, volume 1, pages 39–44, April 1999

8.1 Challenges when using Fitted Q-Iteration

Unfortunately, while in the tabular case (maintaining a value for each state-action pair) the Q-function function converges [11] as the number of iterations of value-iteration (or policy-iteration) steps increases to ∞, this does not generically hold under function approximation. The value function might converge, diverge, oscillate, or behave chaotically. Perhaps worse, meaningful bounds on the resulting performance of a policy learned using value function approximation can be either hard to obtain or vacuous.

[11] Under suitable assumptions discussed earlier.

Fitted Q-iteration and Fitted Value Iteration (a similar algorithm as fitted Q-iteration but approximates the value function and counts on a model to find optimal controls), especially the infinite horizon version, is prone to bootstrapping issues in the sense that sometimes it does not converge. Since these methods rely on approximating the value function inductively, errors in approximation are propagated, and, even worse, *amplified* as the algorithm encourages actions that lead to states with sub-optimal values.

The key reason behind this is the minimization operation performed when generating the target value used for the action value function. The minimization operation pushes the desired policy to visit states where the value function approximation is less than the true value of that state– that is to say, states that look more attractive than they should. This typically happens in areas of state spaces which are very few training samples and could, in fact, be quite bad places to arrive. From a learning theory perspective,

this can be viewed as a violation of the i.i.d assumption on training and test samples.

The following examples from [3] [Boyan and Moore, 1995] demonstrate this problem [12].

[12] All figures from Boyan et. al

Example: 2D gridworld

Figure 8.1.1 shows the 2D grid world example, which has a linear true value function J^*.

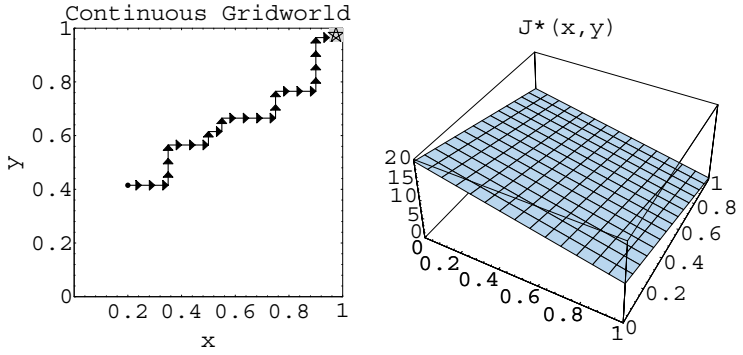


Figure 8.1.1: The continuous gridworld domain.

Figure 8.1.2 shows that VI with converges to the true value function.

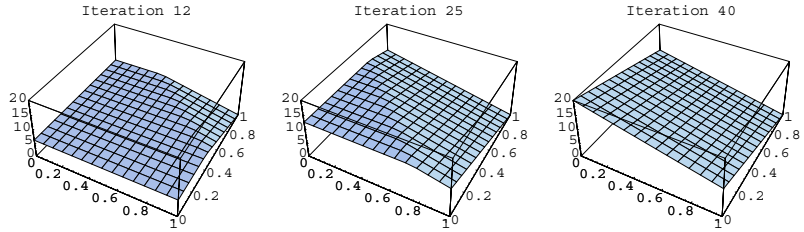


Figure 8.1.2: Training with discrete value iteration.

However, figure 8.1.3 shows that Fitted Value Iteration with quadratic regression fails to converge. The quadratic function, in trying to both be flat in the middle of state space and bend down toward 0 at the goal corner, must compensate by underestimating the values at the corner opposite the goal. These underestimates then enlarge on each iteration, as the one-step lookaheads indicate that points can lower their expected cost-to-go by stepping farther away from the goal.

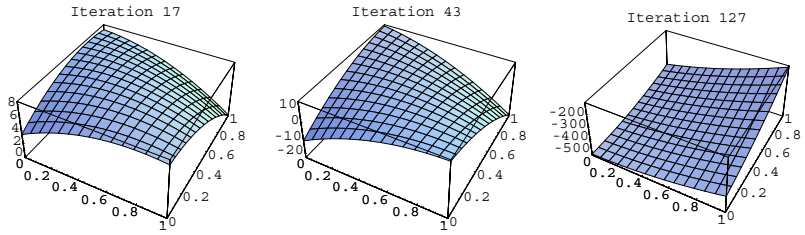


Figure 8.1.3: Training with quadratic regression. The value function diverges. Fitted Value Iteration with quadratic regression underestimates the values at the corner opposite the goal, and these underestimates amplify at each iteration.

Example: car on hill

Figure 8.1.4 shows the car-on-hill example.

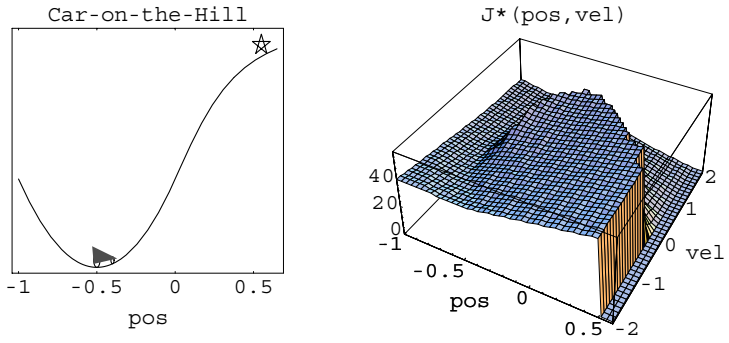


Figure 8.1.4: The car-on-the hill domain.

Figure 8.1.5 shows that a two layer MLP can also diverge to underestimate the costs.

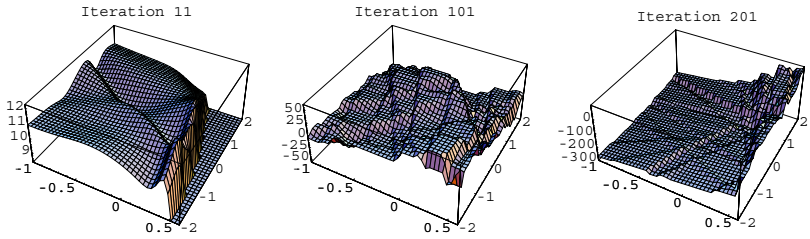


Figure 8.1.5: Training with neural network.

8.2 *Approximate Policy Iteration*

In the previous section we looked at how approximating the action-value function can potentially be effective in large state spaces. In this section, we'll consider approximating the action-value function for a policy from a batch of offline data and then improving that policy. The process of evaluating a policy will be more stable compared with fitted Q iteration as the min operation will no longer be used in the training loop. As with policy iteration, there are two fundamental steps involved in approximate policy iteration process - policy evaluation and policy improvement. We'll consider how *trust-region* and *line search techniques* can control the change of distribution problems that results when we update the policy later in later lectures.

Policy Evaluation

Algorithm 12: Approximate policy evaluation with finite horizon

Algorithm `Evaluate`$(\{x_i, a_i, c_i, x'_i\}_{i=1}^n, T)$
- $Q^\pi(x, a, T) \leftarrow 0,\ \forall x \in \mathbb{X}, a \in \mathbb{A}$
- **forall** $t \in [T-1, T-2, \ldots, 0]$ **do**
  - $D^+ \leftarrow \emptyset$
  - **forall** $i \in 1, \ldots, n$ **do**
    - input $\leftarrow \{x_i, a_i\}$
    - target $\leftarrow c_i + \gamma Q^\pi(x'_i, \pi(x'_i, t+1), t+1)$
    - $D^+ \leftarrow D^+ \cup \{\text{input}, \text{target}\}$
  - **end**
  - $Q^\pi(\cdot, \cdot, t) \leftarrow \textsc{Learn}(D^+)$
- **end**
- **return** $Q^\pi(\cdot, \cdot, 0 : T-1)$

In Algorithm 12, improved stability of the function approximation comes from that fact that we are interested in approximating Q^π and not Q^*. This kind of stability often turns out to be critical, and many practical RL implementations favor a policy iteration variant. Naturally, there is also a "batch"

infinite-horizon version of the above algorithm:

Algorithm 13: Approximate policy evaluation with infinite horizon

Algorithm `Evaluate`$(\{x_i, a_i, c_i, x_i'\}_{i=1}^n, \pi)$
  $Q^\pi(x,a) \leftarrow 0,\ \forall x \in \mathbb{X}, a \in \mathbb{A}$
  while *not converged* **do**
    $D^+ \leftarrow \emptyset$
    forall $i \in 1, \ldots, n$ **do**
      input $\leftarrow \{x_i, a_i\}$
      target $\leftarrow c_i + \gamma Q^\pi(x_i', \pi(x_i'))$
      $D^+ \leftarrow D^+ \cup \{\text{input}, \text{target}\}$
    end
    $Q^\pi \leftarrow \textsc{Update}(Q^\pi, D^+)$
  end
  return Q^π

Function approximation induces very significant problems in computing good policies or value functions. Lets take a closer look at the problems that result.

Function Approximation Divergence

We consider now the more stable variant–function approximation of the policy evaluation step alone–rather than the more complex (non-linear) Q-iteration variant. [13] Even here, Tsitsiklis and Van Roy [6] demonstrate that without care, function approximation has the potential to behave very poorly.

[13] Below we'll discuss that the more difficult to manage problems come from the changing the policy.

Consider the MDP in Figure 8.2.1 has two states S1 and S2. The following details the setup:

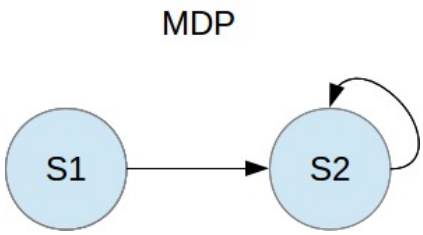


Figure 8.2.1: Two state MDP

1. The reward for being at any state (hence the true value function) is $\{0, 0\}$
2. Consider a discount factor $\gamma = 0.9$
3. The feature $\{x\}$ is simply the numerical value of the state $\{1, 2\}$
4. The value function is approximated with linear function: $V(s) \leftarrow w^T x$

The graphical view of the value function approximation is shown in Figure 8.2.2. Since the reward is always 0, we know the true value function is $\{0, 0\}$. This corresponds to $w = 0$. We will now examine if the approximation converges to this value.

Let's start with $w = 1$. One round of value iteration yields the following

target values for the function approximator

$$V_\pi(s) = r(s, \pi(s)) + \gamma V(s')$$
$$V(s1) \leftarrow 0 + \gamma w * 2 = 1.8$$
$$V(s2) \leftarrow 0 + \gamma w * 2 = 1.8$$

If a least squares approach is used to fit to this data, we'd arrive at $w = 1.2$. Repeated iteration eventually results in the function approximator blowing up exponentially in the number of iterations iterations/number of backups that are performed. [14]

[14] One might hope that the finite horizon variant might not suffer from divergence in this example. That is technically correct (observed by Wen Sun), but is ineffective as the error instead simply grows exponentially in horizon length).

Some Remedies for Divergence

If the training data is weighted by how much time the agent visits a state, then divergence problem can be arrested *for linear function approximators*. In our example, if we spend $t = 1$ time-steps in S1, then we spend $\frac{\gamma}{1-\gamma} = 9$ time-steps at S2. If this is used as a weight in the weighted least squares fitting, then after the first iteration $w = 0.92$, i.e, it proceeds towards the correct value 0. This *on-policy* weighting, where the loss is weighted by the time spent in each state can be demonstrated to ensure convergence. Unfortunately, the same result does not hold for a more general class of function approximators. [6] An entire literature has grown up around attempts to maintain the advantages of approximating the dynamic programming iterations while ensuring convergence in more general settings. Sutton and Barto's book [15] extensively covers these efforts and is highly recommended.

[15] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998

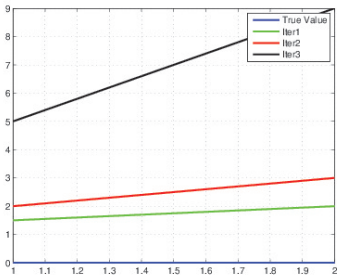


Figure 8.2.2: Approximate Value Function Iteration

Policy Improvement

The second step of the Approximate Policy Iteration process is to update or *improve* the policy. We select a new policy by simply acting greedily with respect to the estimated Q-function of the old one:

$$\pi'(x, t) = \arg\min_a Q^\pi(x, a, t) \tag{8.2.1}$$

In API we have moved the dominant form of instability to *this* step of the process.

The Central Problem of Approximate Dynamic Programming

We discussed before the problem of value function approximation overestimating how good it thinks a state is, and then this error amplifying as

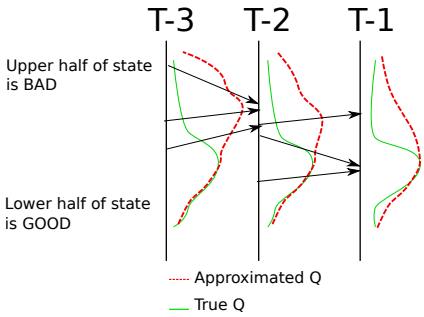


Figure 8.2.3: Value function overestimation in value iteration

Bellman backups proceed. Figure 8.2.3 shows an illustration of this effect. Because the upper half of the state space (which is bad) is overestimated by the function approximator, policies switch to direct probability mass towards that state by choosing actions that make arriving at these states more likely. Error in overestimation of the value function has a cascading effect as we iterate backwards in time.

We further noted that the pure policy evaluation variant of dynamic programming is much more stable– without the *max* to drive behavior towards states with high value estimates we are less subject to the amplification of errors. However, on the surface it seems that we've merely pushed the problem into the *policy improvement* step. That is, while the estimation of the action-value function for a current policy becomes stable, the improvement step would instead drive probability mass towards states-actions that tend to be over-estimates of quality, leading to instability between iterations of any approximate policy iteration procedure.

This objection is, in fact, well-founded and approximate policy iteration algorithms aren't noted to be more stable or effective than approximate value iteration counterparts. However, the maintenance of an explicit policy opens up a new possibility: the ability to manage or mitigate the distribution shift that occurs when we update the policy.

Conservativity and Trust Regions

A broad class of algorithms, initiated by the seminal development of *Conservative Policy Iteration (CPI)* [16] constrain modification to the current policy to prevent the state-action distribution from changing too radically between iterations and thus ensure errors don't explode. The result is algorithms that are stable and effective, although they can be slower than raw policy iteration. CPI modifies the policy update step to *stochastically mix* [17] between policies $\pi^{\text{new}} = \alpha\pi^{\text{new greedy}} + (1-\alpha)\pi^{\text{old}}$, where the mixing weight α is interpreted as the probability of choosing that component. Careful analysis in [18] ensures a strategy for choosing α that ensures improvement, while in practice a simple *line-search* strategy can be employed to ensure monotonic improvement.

This is a somewhat impractical algorithm as it can take many steps and requires maintenance of a mixture of a number of policies equal to the number of update steps. Later approaches, including *No-Regret Policy Iteration* [19] and the Natural or Covariant Policy Search Approach [20] (and later implementations of these like "Trust Region Policy Optimization" [21]) manage to keep one policy, albeit a typically stochastic one, but keep the same intuition of a controlled policy change through the stability of no-regret learning, line

[16] S. Kakade and J. Langford. Approximately optimal approximate reinforcement learning. In *Proceedings of the 19th International Conference on Machine Learning (ICML)*, 2002

[17] That is to say, choose with that probability at each time-step of execution of the policy.

[18] S. Kakade and J. Langford. Approximately optimal approximate reinforcement learning. In *Proceedings of the 19th International Conference on Machine Learning (ICML)*, 2002

[19] S. Ross and J. A. Bagnell. Reinforcement and imitation learning via interactive no-regret learning. *arXiv preprint arXiv:1406.5979*, 2014

[20] ; and J. A. Bagnell, A. Y. Ng, S. Kakade, and J. Schneider. Policy search by dynamic programming. In *Advances in Neural Information Processing Systems*, 2003

[21]

search or trust-region constraints. We defer discussion of these methods to the *Policy Search* chapter **??**.

Interestingly, there becomes no clear line between the modern, controlled *Approximate Policy Iteration* algorithms and algorithms that are variants of *Policy Gradient*. When an action-value function estimator is used that "bootstraps" using Bellman updates, we tend to view them as *API* algorithms. When the updates are made using pure roll-out estimates, we tend to view them as "policy gradient" algorithms. In practice, the distinction in practical use of the terms is somewhat artificial.

8.3 Policy Search by Dynamic Programming

Our focus thus far has been on Bellman *bootstrapping* and approximating the value function either of a given, or optimal, policy. Can we use the core idea of dynamic programming *without* bootstrapping values? Richard Bellman's thoughts shed some light on this issue:

> "An optimal policy has the property that whatever the initial decision may be, the remaining decisions constitute an optimal policy .... [for the resulting state]"

The central idea of dynamic programming is that it does not matter how a decision maker arrives at a state; rather, what matters is that given arrival at the state, the decision maker chooses optimally thereafter. This insight allows us to solve problems recursively. This notion is related to the monotonic improvement of policy iteration. If we *cache policies* and re-estimate the value function at every iteration backwards in time, we avoid the over-estimation and compounding errors problem discussed above as we get unbiased estimates of the real costs that will occur in the future, and errors are not amplified as we proceed through iterations.

Let's try and make this intuition about caching policies concrete. As is standard in dynamic programming, we proceed backwards (over a finite horizon) from $T-1$. At iteration $T-\tau$, instead of memoizing (approximately) a value function in the future and bootstrapping from that, we memoize just the policies in the future and "roll-out" the total cost of an action and future policy decisions all the way to $T-1$. A new policy is learned via estimating an action-value function at $T-\tau$. [22] for a single time step given the rollouts. A new policy is installed at the time-step $T-\tau$. Let's walk through how this works below.

[22] Or, often more powerfully, simply optimizing the policy directly to choose actions with high future returns.

A Sketch of an Algorithm

Let's see what it might look like to use dynamic programming *without* memoizing values:

Time T − 1:

We can approximate $\tilde{\pi}^{*,T-1}(x) = \arg\min_a c(x,a)$ either analytically or via sampled states from a (for now fixed) distribution $\mu_{T-1}(s)$ which we'll call the *baseline distribution*. We'll assume for now that actions are simply chosen uniformly at random [23]. This forms our approximation of the optimal policy at $T-1$.

[23] Or that all of them are tried instead for a given state! That has lower variance, but requires a reset access model that lets us return to the exact same state.

Time T − 2:

For any sampled input pair $\{x_i, a_i\}$, the target value is $c(x_i, a_i) + c(x', \pi^{x',T-1})$. So an error in approximation of π does not bootstrap, it shows up as the policy is always evaluated honestly. However, we again need to specify a distribution of states to optimize with respect to, $\mu_{T-2}(s)$, and given these samples can attempt to find a one-step optimal policy $\tilde{\pi}^{*,T-2}(x)$ that minimizes the average cost under the distribution of samples.

Similarly now for any k, (starting with k=2 and moving backwards in time) we can compute: **Time T − k:**

For a sampled input pair $\{x_i, a_\}$, the target value is $c(x_i, a_i) + c(x', \pi^{x',T-k+1}) + c(x'', \pi^{x',T-k+2} + \cdots$.

Note that this approach address the problem of learning over an *exponentially* large set of policies, but it does with a *quadratic* in horizon T dependence, rather than the *linear* in T dependence that is achieved by policy or value iteration. [24]

[24] Of course, whether this is cost is "worth" it or not depends on both the horizon length and how much errors are amplified by backups.

The Baseline distribution

Note that the algorithm requires a distribution $\mu_t(s)$ from which to draw sample states (as do any of the batch fitted iteration methods). This presents something of a *chicken-or-the-egg* situation as intuitively (and which we'll quantify below) we'd like to sample states from where the optimal policy would visit. This kind of requirement of having an idea of where to sample states is fairly common though: The PSDP approach was partially inspired by the work of [25] and by differential dynamic programming (DDP) generally where policies are generated using as input an initial sample of trajectories. An insight provided by that paper is the usefulness of having information regarding where good policies spend time. This can come from a heuristic initial policy or demonstration by an "expert" at a task. The idea of *baseline distribution* is the natural probabilistic generalization of an *initial trajectory*.

[25]

In essence, the baseline distribution tells our learners where to focus their effort, and shortcuts the the difficulties of *global exploration*. We count on the reset access model and the baseline distribution to solve the hard problem of identifying and getting to states that really matter. We'll spend more time in later lectures discussing baseline distributions and exploration as the general idea of leveraging a baseline distribution is a powerful "cheat" that is equally applicable to policy search/gradient methods as it is to the dynamic programming ones mentioned here.

PSDP as classification

With the crude sketch of a meta-algorithm in hand, we can consider a natural instantiation of PSDP using calls to a supervised learning classification

algorithm.

Algorithm 14: PSDP using classification

Data: Given weighted classification algorithm C and baseline distribution $\mu_t(s)$.
foreach $t \in T-1, T-2, \ldots, 0$ **do**
  Sample a set of n states s_i according to μ_t:
  foreach s_i **do**
    foreach *a in* $\mathcal{A}$ **do**
      Estimate $Q_{a,\pi_{t+1}\ldots,\pi_{T-1}}(s_i)$ by rolling out a trajectory of length $T-t$ starting from s_i and using the action a on the generative model. At each time step after the initial one use the previously computed (near-optimal) policies to choose actions.
    Compute (dis)advantages: For each sampled state s_i, compute the best (highest value) action and denote it a_i^*.
  Create a training set L consisting of tuples of size $|\mathcal{A}|n$: $\langle s_i, a, Q_{a_i^*,\pi_{t+1}\ldots,\pi_{T-1}}(s_i) - Q_{a,\pi_{t+1}\ldots,\pi_{T-1}}(s_i)\rangle$
  Set $\pi_t = C(L)$, where the classifier C is attempting to minimize the weighted 0/1 loss, or an appropriate surrogate.

In the algorithm above we have replaced idealized expectations with Monte Carlo estimates and what it naturally an optimization over one-step policies by a call to an arbitrary supervised-learning algorithm. If we can perform the supervised learning task at each step well we will achieve good performance at the RL task *at least relative to the baseline distribution*.

Action-value approximation via regression. A particular variant of the sample based PSDP above can be implemented if it is possible to efficiently find an approximate action-value function $\tilde{Q}_{a,\pi_{t+1}\ldots,\pi_{T-1}}(s)$, *i.e.*, if at each time-step we can ensure that $\epsilon \geq \mathrm{E}_{s\sim\mu_t(s)}[\max_{a\in A}|\tilde{Q}_{a,\pi_{t+1}\ldots,\pi_{T-1}}(s) - Q_{a,\pi_{t+1}\ldots,\pi_{T-1}}(s)|]$.

(Recall that the policy sequence $(a, \pi_{t+1}\ldots,\pi_{T-1})$ always begins by taking action a.) If the policy π_t is greedy with respect to the estimated action value $\tilde{Q}_{a,\pi_{t+1}\ldots,\pi_{T-1}}(s)$, then we can show that this induces a policy that is within $2T\epsilon$ [26] of choosing the optimal action according to the distribution μ. It is important to note that this error is phrased in terms of an *average error* over state-space, as opposed to the worst case errors over the state space that are more standard in dynamic programming algorithms and drive the instabilities of those approaches. We can intuitively grasp this by observing that value iteration style algorithms may amplify any small error in the value function by pushing more probability mass through where these errors are. PSDP, however, as it does not use value function backups, cannot make this same error; the use of the computed policies in the future keeps it honest. There are numerous efficient regression algorithms that can minimize this, or approximations to it.

[26] Think about where the 2 comes from here!

"Convergence" and Partial Observability

Note that even as the time horizon we consider gets very large when we are in the function approximation setting, Q does **not** necessarily converge as

$k \to T$ even when it would in the full tabular setting. To see why this might be so, consider an extreme example.

Imagine a hypothetical situation of making a two legged robot walk. Further, imagine we limit our policy to have only a single feature given to the function approximation: the time-step t — *i.e.* no description of state whatever. As demonstrated in [27], an algorithm like PSDP can actually learn a sequence of open loop torques that make the robot perform an effective, albeit brittle, walking motion. The value of choosing some action will be *very different* even at neighboring time-steps, of course, because we're encoding an open-loop strategy here. Generically, this is true: we can get different Q-functions at neighboring time steps– this is a strong indication, however, of aliasing of real underlying states.

[27] J. A. Bagnell and J. Schneider. Covariant policy search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2003

Algorithm 15: Iterated Policy Search by Dynamic Programming

Algorithm `Iterated-PSDP(`π`)`

Start with arbitrary time-varying policy π_0
$k \leftarrow 0$
while *not converged* **do**
  for $\forall x \in \mathbb{X}, t \in \{0, \ldots, T-1\}$ **do**
    $\pi_{k+1}(x,t) \leftarrow \operatorname{argmin}_{a \in \mathbb{A}} Q^{\pi_k}(x,a,t)$ Collect samples $\{\{x_t^{(i)}, a_t^{(i)}, c_t^{(i)}, x_{t+1}^{(i)}\}_{t=0}^{T-1}\}_{i=1}^{n}$ by executing policy π_k
    $Q^{\pi_k+1} \leftarrow$ `Learn(`$\{\{x_t^{(i)}, a_t^{(i)}, c_t^{(i)}, x_{t+1}^{(i)}\}_{t=0}^{T-1}\}_{i=1}^{n}, \pi_k$`)`
  end
  $k \leftarrow k+1$
end
return $\pi_k(x), \forall x$

Understanding Performance Guarantees

There are multiple possible bounds that can be established for the algorithms above. Perhaps the most insightful on is a bound on performance with a multiplicative dependence on average error in the case that we are willing to compare ourselves against *all* policies rather than expressing regret with respect to a limited class. [28] We state the bound as follows:

[28] Although the search in PSDP may, of course, still be conducted over the limited class.

Theorem 2 (MDP Performance Bound)**.** *Let $\pi = (\pi_0, \ldots, \pi_{T-1})$ be a non-stationary policy returned by an ε-approximate version of PSDP in which, on each step, the policy π_t found comes within ε of maximizing the value over* all *policies. I.e.,*

$$\mathrm{E}_{s \sim \mu_t}[V_{\pi_t, \pi_{t+1}, \ldots, \pi_{T-1}}(s)] \geq \max_{\pi'} \mathrm{E}_{s \sim \mu_t}[V_{\pi', \pi_{t+1}, \ldots, \pi_{T-1}}(s)] - \varepsilon\,. \tag{8.3.1}$$

Then for all possible policies (including including the optimal one) π_{ref} and its induced distribution over states $\mu_{\pi_{\mathrm{ref}}}$ we have that

$$V_\pi(s_0) \geq V_{\pi_{\mathrm{ref}}}(s_0) - \sum_t \varepsilon || \frac{\mu^t_{\pi_{\mathrm{ref}}}}{\mu^t} ||_\infty$$

where the infinity norm refers to the sup *of state space.*

We sketch a proof here. It is cleanest to apply the *Performance Difference Lemma* we developed earlier, but there is a certain value to understanding the induction that is being applied directly. [29] **Proof:** It is enough to show that for all $t \in T-1, T-2, \ldots, 0$,

[29] Exercise: Apply the performance difference lemma for a simplified version of this proof.

$$\mathrm{E}_{s\sim\mu^t_{\pi_{\mathrm{ref}}}}[V_{\pi_{\mathrm{ref}}{}^t,\ldots}(s) - V_{\pi^t,\ldots}(s)] \leq \sum_{\tau=t}^{T-1} \varepsilon ||\frac{\mu^\tau_{\pi_{\mathrm{ref}}}}{\mu^\tau}||_\infty$$

This again follows by induction, the inductive step being the non-trivial part:

$$\begin{aligned}
& \mathrm{E}_{s\sim\mu^t_{\pi_{\mathrm{ref}}}}[V_{\pi_{\mathrm{ref}}{}^t,\ldots}(s) - V_{\pi^t,\ldots}(s)] \qquad (8.3.2)\\
= \;& \mathrm{E}_{s\sim\mu^t_{\pi_{\mathrm{ref}}}}[V_{\pi_{\mathrm{ref}}{}^t,\ldots}(s) - V_{\pi_{\mathrm{ref}}{}^t,\pi^{t+1}\ldots}(s)] + \mathrm{E}_{s\sim\mu^t_{\pi_{\mathrm{ref}}}}[V_{\pi_{\mathrm{ref}}{}^t,\pi^{t+1}\ldots}(s) - V_{\pi^t,\ldots}(s)]\\
\leq \;& \mathrm{E}_{s\sim\mu^{t+1}_{\pi_{\mathrm{ref}}}}[V_{\pi_{\mathrm{ref}}{}^{t+1},\ldots}(s) - V_{\pi^{t+1}\ldots}(s)] + \mathrm{E}_{s\sim\mu^t_{\pi_{\mathrm{ref}}}}[\max_{a_t} Q_{a^t,\pi^{t+1}\ldots}(s) - V_{\pi^t,\ldots}(s)]\\
\leq \;& \mathrm{E}_{s\sim\mu^{t+1}_{\pi_{\mathrm{ref}}}}[V_{\pi_{\mathrm{ref}}{}^{t+1},\ldots}(s) - V_{\pi^{t+1}\ldots}(s)] + \mathrm{E}_{s\sim\mu^t_{\pi}}[\max_{a_t} Q_{a^t,\pi^{t+1}\ldots}(s) - V_{\pi^t,\ldots}(s)]||\frac{\mu^t_{\pi_{\mathrm{ref}}}}{\mu^t}||_\infty\\
\leq \;& \mathrm{E}_{s\sim\mu^t_{\pi}}[V_{\pi_{\mathrm{ref}}{}^t,\pi^{t+1}\ldots}(s) - V_{\pi^t,\ldots}(s)] + \varepsilon||\frac{\mu^t_{\pi_{\mathrm{ref}}}}{\mu^t}||_\infty\\
= \;& \sum_{\tau=t}^{T-1} \varepsilon||\frac{\mu^\tau_{\pi_{\mathrm{ref}}}}{\mu^\tau}||_\infty
\end{aligned}$$

■

We are able to "change measures" with the infinity norm (i.e., the largest value taken at any state) of the ratio of the probability distributions here because the action-value function optimized over a is **always** greater than the value of any other policy. This change of measure follows directly by multiplying inside the second expectation by $\frac{\mu_t}{\mu_t}$, and then bounding the result.

This bound is powerful in that it lets our error go to zero even if we do not get a perfect distribution μ_t as long as we drive our expected error ε to be low. We can also drop the dependence of μ_t on t, by simply averaging all the time slice distributions together

$$\frac{1}{T}\sum_{t=0}^{T-1} \mu_t$$

if we are willing to learn our policies to ε/T error.

This bound provides insight by indicating that it is very important that the "training" distribution be close in a particular way to the distribution induced by any policy (notably the optimal one) we want to compete against. In particular, when training each classifier we want to ensure that we put mass on all places where a good policy spends time so that the ratio $\frac{\mu^\tau_{\pi_{\mathrm{ref}}}}{\mu^\tau}$ is never too large. This makes intuitive sense– we're better to make sure our learner has seen examples of possible situations it can get into even if this means removing some of the mass from more probable instances. Crudely speaking, in choosing a μ we should err on the side of "smearing" our best guess of the distribution $\mu_{\pi_{\mathrm{ref}}}$ induced by the optimal policy across neighboring states.

This style of proof where we use some variation on the performance difference lemma and a change of distribution is very common to the analysis of almost all modern approximate policy iteration methods.

Iterated PSDP

PSDP as presented takes as input the set of space-time distributions μ_t and generates a policy in polynomial time with non-trivial *global* performance guarantees with respect to μ_t. In many applications, we are able to provide a useful state-time distribution to the algorithm. For instance, in control and decision tasks human performance can often provide a baseline distribution to initialize PSDP. We also often have heuristic policies that can be used to initialize the algorithm. Finally, domain knowledge often provides useful indications of where good policies spend time.

In any of these cases, we do not have an accurate estimate of μ for an optimal policy. A natural approach is to apply PSDP as the inner loop for a broader algorithm that attempts to simultaneously compute μs and uses PSDP to compute optimal policies with respect to it. Perhaps the most natural such algorithm is given below.

Algorithm 16: IteratedPSDP(μ, π, v)

Let π_{new} = PSDP(μ)
v_{new} = Value(π_{new})
μ_{new} = ComputeInduced$\mu(\pi_{new})$ **in**
if $v_{new} \leq v$ **then**
  return π

else
  return IteratedPSDP ($\mu_{new}, \pi_{new}, v_{new}$)

Value here is a function that returns the performance of the policy and ComputeInducedμ returns a new baseline distribution corresponding to a policy. These can both be implemented a number of ways, perhaps the most important being by Monte-Carlo sampling.

We start Iterated PSDP with $v = 0$ and a null policy in addition to our "best-guess" μ. Iterated PSDP can be seen as a kind of search where the inner loop is done optimally. For exact PSDP ($\varepsilon = 0$) on an MDP with finite states and actions, we can prove that performance improves with each loop of the algorithm and converges in a finite number of iterations.

In the case of approximation, it is less clear what guarantees we can make. Performance improvement occurs as long as we can learn policies at each step that have smaller average residual advantages than the policy we are attempting to improve over.

Summary

PSDP is a useful algorithm template and can serve as a kind of design pattern for approximate DP algorithms and as a tool of theoretical ananlsis. While there are a number of practical applications of PSDP, even within robotics[30], it is not nearly as commonly used in practice as online variants of the approximate dynamic programming or policy gradient algorithms we'll investigate later.

[30]

8.4 Related Reading

[1] Ernst, Damien, Pierre Geurts, and Louis Wehenkel, *Tree-based batch mode reinforcement learning*. Journal of Machine Learning Research 2005.

[2] Baird, L. (1995). Residual algorithms: Reinforcement learning with function approximation. In Machine Learning Proceedings 1995 (pp. 30-37). Morgan Kaufmann.

[3] Boyan, Justin A and Moore, Andrew W, *Generalization in Reinforcement Learning: Safely Approximating the Value Function*. NIPS 1994.

[4] Gordon, Geoffrey J, *Stable function approximation in dynamic programming*. DTIC Document 1995.

[5] Bagnell, J. A., Kakade, S. M., Schneider, J. G., and Ng, A. Y. (2004). Policy search by dynamic programming. NIPS, 2004.

[6] J. N. Tsitsiklis and B. Van Roy, *An Analysis of Temporal-Difference Learning with Function Approximation*, IEEE Transactions on Automatic Control, Vol. 42, No. 5, 1997.

9
Temporal Difference Learning and Q-Learning

In the previous chapter, we covered several sample-based reinforcement learning algorithms including Fitted Q-Iteration and Approximate Policy Iteration. These methods are sometimes called *batch methods* or *offline methods* because a batch of samples is collected and a fitted value function (or action-value function) is found by minimizing the training error for these samples in one "chunk". Offline methods like these make efficient use of available training data, but are computationally expensive and suffer from high memory consumption as the number of samples increases. They also tend to suffer quite badly from the distribution mismatch problems we described in the last chapter.

In this lecture, we present several *online* techniques that perform an incremental update to a value function estimate after each state transition (x, a, r, x'). [1] Such online methods can learn a policy with relatively low computational and memory cost because the updates are made based on a single state transition. Such a state transition (x, a, r, x') may be referred to in the literature as an *experience*.

[1] Note that in this chapter we have switched from cost to reward r, as is common in the reinforcement learning literature.

First, we present the *Temporal-Difference (TD) method* for online policy evaluation. Next, we present a TD-variant denoted *SARSA* that extends online policy evaluation to the action-value function. Finally, we explore *Q-learning* as a method for finding the action-value function for the optimal policy, and hence finding the optimal policy.

In this lecture, we consider the infinite time-horizon setting, and for notational simplicity will usually assume a deterministic policy. Algorithms generalize to stochastic polices aswell. Recall that the value function for a fixed policy π satisfies the Bellman equation:

$$V^{\pi}(x) = E\left[\sum_{t=0}^{\infty} \gamma^t \, r(x_t, \pi(x_t))\right], \quad \text{where } x_0 = x. \tag{9.0.1}$$

The action-value function for a fixed policy π satisfies:

$$Q^{\pi}(x, a) = r(x, a) + E\left[\sum_{t=1}^{\infty} \gamma^t \, r(x_t, \pi(x_t))\right], \quad \text{where } x_0 = x. \tag{9.0.2}$$

The Bellman equation in this case,

$$\begin{aligned} V^{\pi}(x) &= r(x, \pi(x)) + \gamma \, \mathbb{E}_{p(x'|x,\pi(x))}[V^{\pi}(x')] \\ Q^{\pi}(x, a) &= r(x, a) + \gamma \, \mathbb{E}_{p(x'|x,a)}[Q^{\pi}(x', \pi(x'))]. \end{aligned} \tag{9.0.3}$$

The Bellman equations for the optimal value function V^* and action-value function Q^* of the optimal policy π^* are naturally,

$$\begin{aligned} V^*(x) &= \max_{a' \in \mathbb{A}} \left(r(x,a) + \gamma \, \mathbb{E}_{p(x'|x,a)}[V^*(x')] \right) \\ Q^*(x,a) &= r(x,a) + \gamma \, \mathbb{E}_{p(x'|x,a)}[\max_{a' \in \mathbb{A}} Q^*(x',a')]. \end{aligned} \tag{9.0.4}$$

9.1 *Temporal-Difference Learning*

Temporal-difference (TD) Learning is an online method for estimating the value function for a fixed policy π. The principle idea behind TD-learning is that we can learn about the value function from *every* experience (x, a, r, x') as a robot traverses the environment rather than only at the end of a trajectory or trial. [2]

Given an estimate of the value function $\tilde{V}^\pi(x)$ we would like to perform an update in order to minimize the squared loss,[3],

$$\mathcal{L} = \frac{1}{2} \left(V^\pi(x) - \tilde{V}^\pi(x) \right)^2 . \tag{9.1.1}$$

Since we do not yet know the value function, evaluating this loss requires evaluating equation (9.0.1). Naïvely, this method would require waiting until the end of an episode before updating $\tilde{V}^\pi(x)$. Instead, we estimate $V^\pi(x)$ as $y = r + \gamma \, \tilde{V}^\pi(x')$ and perform an online update for each experience $(x, \pi(x), r, x')$. Plugging this estimate into the loss function we get [4]

$$\mathcal{L}_{\text{approx}}(y, \tilde{V}^\pi) = \frac{1}{2} \left(y - \tilde{V}^\pi(x) \right)^2 . \tag{9.1.2}$$

The partial derivative of eq. (9.1.2) with respect to $\tilde{V}^\pi$ is:

$$\nabla_{\tilde{V}^\pi(x)} \mathcal{L}_{\text{approx}} = \left(y - \tilde{V}^\pi(x) \right) . \tag{9.1.3}$$

If we assume now a parametric form for $\tilde{V}^\pi(x)$ in terms of parameters θ, we can then use the chain rule we can then express $\nabla_\theta \mathcal{L} = \left(y - \tilde{V}^\pi(x) \right) \nabla_\theta \tilde{V}^\pi(x)$

In the case of a tabular representation (one value for each state), our update rule with a step-size of α would simply be:

$$\begin{aligned} \tilde{V}^\pi(x) &\leftarrow \tilde{V}^\pi(x) + \alpha \left(r + \gamma \tilde{V}^\pi(x') - \tilde{V}^\pi(x) \right) \\ &\leftarrow (1 - \alpha) \tilde{V}^\pi(x) + \alpha \left(r + \gamma \tilde{V}^\pi(x') \right) . \end{aligned} \tag{9.1.4}$$

The term $\left(r + \gamma \tilde{V}^\pi(x') - \tilde{V}^\pi(x) \right)$ is known as the *TD error*.

By looking at the second line of (9.1.4), one may notice that TD-learning is also closely related to an *exponential moving average*.

[2] TD is truly one of the core algorithmic ideas in RL. It forms the heart of *TD-Gammon*, the first algorithm to beat humans at the difficult stochastic game of backgammon. That paper is a masterpiece and set the pattern for modern self-play RL in games. 's outstanding book provides **much** more details on Temporal Difference methods and is highly recommended.

; and R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998

[3] Technically, this squared loss is an estimate of the Bellman error $E_{d^\pi(x)}[\frac{1}{2} \left(V^\pi(x) - \tilde{V}^\pi(x) \right)^2]$ where $d^\pi(x)$ is the probability of a state x being visit under policy π.

[4] Notice the trick that has been played here! We're treating the value estimate of the future state as if it were "correct"– as if it were not a function of the parameters that define our value function. This is, of course, totally incorrect. The *Bellman residual* and the Residual Gradient (RG) *is the "obvious" item to optimize and it's gradient and we disc*

Algorithm TD

The TD-learning algorithm is shown in Algorithm 17.

Algorithm 17: The TD-learning algorithm.

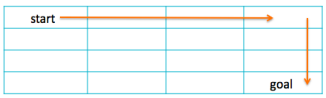
Initialize $\tilde{V}^\pi$
while $\tilde{V}^\pi$ *not converged* **do**
  Initialize x according to a particular starting state
  while x *is not a terminal state* **do**
    apply action $a \leftarrow \pi(x)$
    receive experience $\{x, \pi(x), r, x'\}$
    update $\tilde{V}^\pi(x)$

$$\tilde{V}^\pi(x) \leftarrow (1-\alpha)\tilde{V}^\pi(x) + \alpha\left(r + \gamma\tilde{V}^\pi(x')\right)$$

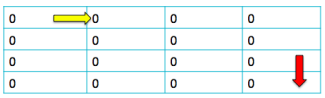
    set $x \leftarrow x'$
return $\tilde{V}^\pi$

Grid-World Example

The diagram below shows a grid-based world, where the robot starts in the upper left $(0,0)$, and the goal is in the lower right $(3,3)$. The robot gets a reward of $+1$ if it reaches the goal, and 0 everywhere else. There is a discount factor of γ. The policy is for the robot to go right until it reaches the wall, and then go down.



We start by initializing $\tilde{V}^\pi(x) = 0,\ \forall x \in \mathbb{X}$.



As the robot moves one cell over from the start state (yellow arrow above), the reward is 0, and the value of both the current state and the next state is 0, so the approximate gradient used in the update rule (9.1.4) evaluates to 0 and no update is performed. As the robot moves into the goal state (red arrow), the reward is 1, so the approximate gradient evaluates to 1. We then update the second-to-last cell with (9.1.4) and we get:

$$\begin{aligned}\tilde{V}^\pi((3,2)) &\leftarrow (1-\alpha)\tilde{V}^\pi((3,2)) + \alpha\left(1 + \gamma\tilde{V}^\pi((3,3))\right)\\ &= (1-\alpha)\times 0 + \alpha \times (1+0) = \alpha.\end{aligned}$$

Another iteration of the algorithm gives us:

0	0	0	0
0	0	0	0
0	0	0	α
0	0	0	0

$$\begin{aligned}
\tilde{V}^{\pi}((3,2)) &\leftarrow (1-\alpha)\tilde{V}^{\pi}((3,2)) + \alpha\left(1+\gamma\tilde{V}^{\pi}((3,3))\right) \\
&= (1-\alpha)\times\alpha + \alpha\times(1+0) \\
&= \alpha + \alpha(1-\alpha), \\
\tilde{V}^{\pi}((3,1)) &\leftarrow (1-\alpha)\tilde{V}^{\pi}((3,1)) + \alpha\left(1+\gamma\tilde{V}^{\pi}((3,2))\right) \\
&= (1-\alpha)\times 0 + \alpha\times(0+\gamma\times\alpha) \\
&= \alpha^2\gamma.
\end{aligned}$$

0	0	0	0
0	0	0	$\alpha^2\gamma$
0	0	0	α + α(1-α)
0	0	0	0

This method is slow, because we have to run the whole policy just to update the next cell. We will see that SARSA and Q-learning has similar issues of inefficient usage of experience.

9.2 *SARSA*

SARSA extends the Temporal-Difference method presented in the previous section to evaluate policies represented by a action-value functions $Q^{\pi}(x,a)$. Similar to the TD case, we wish to evaluate a policy by performing an online update to obtain an estimate, $\tilde{Q}^{\pi}(x,a)$, of the true action-value function $Q^{\pi}(x,a)$:

$$Q^{\pi}(x,a) = r(x,a) + \sum_{t=1}^{\infty}\gamma^t E[r(x_t,\pi(x_t))] \tag{9.2.1}$$

As in TD, we seek to minimize the loss

$$\mathcal{L}_{\text{approx}} = \frac{1}{2}\left(y - \tilde{Q}^{\pi}(x,a)\right)^2 \tag{9.2.2}$$

where $y = r(x,a) + \gamma\tilde{Q}^{\pi}(x',\pi(x'))$. Following a similar derivation as used for the TD update, we arrive at the SARSA update rule:

$$\tilde{Q}^{\pi}(x,a) \leftarrow (1-\alpha)\tilde{Q}^{\pi}(x,a) + \alpha\left[r(x,a) + \gamma\tilde{Q}^{\pi}(x',\pi(x'))\right]. \tag{9.2.3}$$

Algorithm SARSA

The SARSA algorithm is shown in Algorithm 18.

Algorithm 18: The TD-learning algorithm.

Initialize $\tilde{Q}^\pi$
while $\tilde{Q}^\pi$ *not converged* **do**
  Initialize x according to a particular starting state
  while x *is not a terminal state* **do**
    apply action $a \leftarrow \pi(x)$
    receive experience $(x, \pi(x), r, x', \pi(x'))$
    update $\tilde{Q}^\pi(s)$

$$\tilde{Q}^\pi(x,a) \leftarrow (1-\alpha)\tilde{Q}^\pi(x,a) + \alpha\left[r(x,a) + \gamma\tilde{Q}^\pi(x', \pi(x'))\right]$$

    set $x \leftarrow x'$
return $\tilde{V}^\pi$

One may notice that TD-learning and SARSA are essentially approximate policy evaluation algorithms for the current policy. As a result of that they are examples of *on-policy* methods that can only use samples from the *current* policy to update the value and Q function. *Q-learning*, by contrast, is an *off-policy* method that can use samples from *any* policies [5] to update the optimal action-value function.

[5] Although clearly it requires exploring all actions in all states.

9.3 *Q-Learning*

Q-Learning attempts to estimate the *optimal* action-value function $Q^*(x,a)$ from an online stream of experiences. Recall that the Bellman Equation for the optmal action-value function $Q^*(x,a)$ is,

$$Q^*(x,a) = r(x,a) + \gamma\,\mathbb{E}_{p(x'|x,a)}[\max_{a'\in\mathbb{A}} Q^*(x',a')].$$

Suppose we receive experience (x,a,r,x'). If the transition model is deterministic, we could simply update the action-value function as,

$$\tilde{Q}^*(x,a) \leftarrow r + \gamma\max_{a'\in\mathbb{A}}\tilde{Q}^*(x',a').$$

However, just as in SARSA, this performs poorly when the transition or reward functions are stochastic. Instead, we update $\tilde{Q}^*$ to the weighted sum,

$$\tilde{Q}^*(x,a) \leftarrow \alpha\left[r + \gamma\max_{a'\in\mathbb{A}}\tilde{Q}^*(x',a')\right] + (1-\alpha)\tilde{Q}^*(x,a),$$

where $0 \leq \alpha \leq 1$ is the *learning rate*.

One may notice that we do not need the current policy π to update $\tilde{Q}^*$. Moreover, Q-learning approximates the *optimal* action-value function, the Bellman Equation of which does not depend on the specific policy that the agent is executing. Therefore, Q-learning is an *off-policy* algorithm that can use samples from *any* policies to update $\tilde{Q}^*$. From our experience in the last chapter, one should however, be naturally suspicious of any algorithm that claims to be able to this as it must suffer from *distributional shift* as Value or Policy Iteration do.

Q-learning is guaranteed to converge $\tilde{Q}^*$ to the optimal action-value function Q^* as number of iterations $k \to \infty$ given that the following conditions hold:

1. Each state-action pair is visited infinite times
2. $\lim_{k\to\infty} \sum_{k=0}^{\infty} \alpha_k = \infty$
3. $\lim_{k\to\infty} \sum_{k=0}^{\infty} \alpha_k^2 < \infty$,

where α_k is the learning rate at iteration k. The latter two conditions mean that the learning rate α must be annealed over time. Intuitively, this means that the agent begins by quickly updating $\tilde{Q}^*$, then slows down to refine its estimate as it receives more experience.

Fitted Q-Learning

Just as the fitted Q-iteration algorithm, we can use a function approximator to approximate the action-value function.

Suppose that we approximate Q^* with the function Q_θ with parameter θ. Instead of directly updating our action-value function, we now must update θ to achieve the desired change in Q_θ.

To fit θ, we might choose to minimize a loss function

$$\mathcal{L} = \frac{1}{2}\left(y - Q_\theta(x,a)\right)^2$$

that penalizes deviation between the approximate action-value function $Q_\theta(x,a)$ and the value $y = r + \gamma \max_{a' \in A} Q_\theta(x',a')$ predicted by a Bellman backup.

First, we must derive the "gradient" of $\mathcal{L}$. [6] By applying the chain rule, we find

$$\begin{aligned}\nabla_\theta \mathcal{L} &= \left(y - Q_\theta(x,a)\right)\left[\nabla_\theta y - \nabla_\theta Q_\theta(x,a)\right] \\ &= \left(y - Q_\theta(x,a)\right)\left[\gamma \nabla_\theta Q_\theta(x',a^*) - \nabla_\theta Q_\theta(x,a)\right]\end{aligned}$$

where $a^* = \operatorname{argmax}_{a' \in \mathbb{A}} Q_\theta(x',a')$ is the optimal action according to Q^θ. Unfortunately, it is not possible to obtain an unbiased estimate of $Q_\theta(x,a)\nabla_\theta Q_\theta(x',a^*)$ using one sample (x,a,r,x'). We can find the optimal parameter θ by performing gradient descent on $\mathcal{L}$ with the update rule,

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}. \tag{9.3.1}$$

Q-learning, however, assumes that y is constant and approximates the gradient as

$$\widetilde{\nabla}_\theta \mathcal{L} = -\left(y - Q_\theta(x,a)\right)\nabla_\theta Q_\theta(x,a). \tag{9.3.2}$$

The complete fitted Q-learning update rule is found by substituting eq. (9.3.2) into eq. (9.3.1):

$$\begin{aligned}\theta &\leftarrow \theta + \alpha\left[y - Q_\theta(s,a)\right]\nabla Q_\theta(x,a) \\ &\leftarrow \theta + \alpha\left[\left(r + \gamma Q_\theta(x',a^*)\right) - Q_\theta(x,a)\right]\nabla Q_\theta(x,a).\end{aligned}$$

Bellman Residual Method

Fitted Q-learning as described above does *not* implement gradient descent and, thus, is not guaranteed to converge to a local minimum. The *Bellman residual algorithm* avoids the approximation of eq. (9.3.2) by estimating the true gradient $\nabla_\theta \mathcal{L}$.

$$\nabla_\theta \mathcal{L} = \left(y - Q_\theta(x,a)\right)\left(\gamma \nabla_\theta Q_\theta(x',a^*) - \nabla_\theta Q_\theta(x,a)\right).$$

[6] Note again this is the same bogus math where we pretend y is not a function of the parameters. One might naturally ask why not just compute the true gradient? This turns out to be a somewhat nuanced question. One intuitive reason is the notion that our value estimates are likely to better closer to the end of a trial/closer to a goal, and that updates should flow only backwards in time as in dynamic programming updates. If we computed the true gradient (the Bellman residual gradient as it is known , we would have the estimate of the value function in the past changing to more closely match the estimate in the future as well. A further technical difficulty, discussed in Baird's work is that unbiased estimates of the true Bellman residual gradient require **multiple samples** of an action outcome from each state visited. This point of the "derivation" is to give you intuition why you might come up with this rule by thinking about dynamic programming flowing updates backwards and time and the chain rule providing updates.

L. C. Baird. Residual algorithms: Reinforcement learning with function approximation. In *International Conference on Machine Learning*, 1995

This estimation is only unbiased if we can generate two or more independent successor states for taking action a in state s. Generating these samples is trivial if we are able to simulate the system; i.e. have access to a known or learned transition model. If we do not know the transition model, then it is only possible to perform a Bellman residual update if we postpone a backup until the same state-action pair has been observed two or more times. This is often impossible when learning on a real system that has a continuous state-action space.

Exploration Policies

Unlike SARSA, which is an *on-policy* method, Q-learning is an *off-policy* method that can learn from arbitrary (x, a, r, x') experiences, regardless of what policy was used to generate them. This means that it is possible to use an *exploration policy* training that encourages the agent to visit previously unexplored regions of the state-action space. Exploration policies guarantee that the agent visits each state an infinite number of times and ensure convergence when the function is represented by a look-up table.

Two exploration policies that are commonly used with Q-learning are:

1. **ϵ-Greedy**. Choose the greedy action $a = \text{argmax}_{a \in \mathbb{A}} \tilde{Q}(x, a)$ with probability $1 - \epsilon$. Otherwise, with probability ϵ, choose an action uniformly at random $a \sim \text{uniform}(\mathbb{A})$. Higher values of ϵ encourage more exploration. Usually we set ϵ close to 1 as learning starts, and decay $\epsilon \to 0$ as we go along.
2. **Boltzmann Exploration.** Choose action a with probability

$$\pi(a|x) = \frac{\exp\left[\beta \tilde{Q}(x, a)\right]}{\sum_{a' \in \mathbb{A}} \exp\left[\beta \tilde{Q}(x, a')\right]},$$

which is weighted towards selecting actions with higher $\tilde{Q}$-values. Lower values of β encourage more exploration: the exploration policy with $\beta = 0$ is essentially a uniform distribution, as $\beta \to \infty$ the exploration policy becomes the greedy policy

$$\pi(a|x) = \arg\max_{a' \in \mathbb{A}} \tilde{Q}(x, a').$$

Hence, we usually start with β close to 0 and gradually increase β.

9.4 Experience Replay and Replay Buffers

Q-learning and SARSA are computationally efficient, but make inefficient use of data. Unlike batch methods, each sample is only used exactly once. This means that the agent must observe each transition ((x, a, r, x') for Q-learning and (x, a, r, x', a') for SARSA) many times to propagate the reward backwards in time.

Experience relay allows Q-learning to re-use experience multiple times by building a database D of experiences under the currently policy, denoted the *replay buffer*. Once enough data has been collected, the agent performs a fixed number of Q-learning or SARSA updates on the batch. This technique bridges the gap between offline methods and online methods, and can potentially combine the advantages the two.

Moreover, because Q-learning is an off-policy algorithm, the experiences generated from previous trajectories and policies can be *re-used* to update the estimate of action-value functions. Therefore, we can use a replay buffer across Q-learning updates: every time a new experience is generated, it is added to the replay buffer, and the agent performs Q-learning updates using *random samples* from the replay buffer.

A common claim in the literature is that experience relay also helps address the problem of *correlated samples* for fitted Q-learning. In the case of online updates, the a experience is likely to be highly correlated with the previous/next experience because they are from the same trajectory. This makes the function approximator easily *overfit* to the current part of the state space, but fail to perform well for the entire state space. However, such correlation is mitigated when we use a batch of samples from possibly different trajectories to update the function approximator.

9.5 *The Philosophy of Temporal Differences**

This section develops a view of Bellman errors and temporal differences that is less well-studied in the literature. It provides some larger philosophical insight as well as proof techniques, and perhaps will be the root of important technical tools in the future. but can skipped for readers eager to move on to other control approaches.

While the theory of Markov Decision Processes has become a powerful foundation for reasoning about temporal difference algorithms, we might argue that there is a still more fundamental intuition that is being captured in TD-style algorithms. Consider the fully online,*Trace* access model, and the goal of predicting a quantity, like long-term reward, over an arbitrary long time sequence. The fundamental claim at the heart of temporal difference and bellman residual methods is if predictions of long term value are temporarily consistent, then they must also be good proxies for the actual long term reward; equivalently, one cannot make consistent predictions (in the sense of temporal differences) and fail to correctly predict the long term reward.

This notion is quite robust to both noise and imperfect approximation. In fact, we can show still stronger claims: if a learner's predictions compete with the best predictor in a class of learners, then they will also compete with the best in that same class at the goal of long-term value estimation. That is, the errors need not even be small– doing as well as possible at consistency implies doing well at long-term prediction as well. This holds over any possible noise sequence– even adversarial noise, establishing the centrality of the notion of Bellman consistency. The central idea is that methods such as `TD` and `RG` should be fundamentally understood as *online algorithms* as opposed to standard gradient minimization methods, and that one cannot simultaneously make consistent predictions in the sense of `TD` and `BE` while doing a poor job in terms of long-run predictions.

This basic model of relating long-term prediction and temporal differences was established in the work of Schapire and Warmuth [1996]. We follow the analysis of Sun and Bagnell [2015] to provide simple guarantees for a wide class of algorithms.

Problem Setting

Consider a sequence of observations (note they need not have the semantics of state!) that can either be Markovian as we've assumed this far in the lecture, or even adversarially chosen. We define the observation at time step t as $\mathbf{x}_t \in \mathbb{R}^n$, which represents features of the environment at time-step t. Let's assume that feature vector $\mathbf{x}$ is bounded as $\|\mathbf{x}\|_2 \leq X$. The correspond-

ing reward at step t is defined as $r_t \in \mathbb{R}$, where we assume that reward is always bounded $|r| \leq R \in \mathbb{R}^+$. Given a sequence of observations $\{\mathbf{x}_t\}$ and a sequence of rewards $\{r_t\}$, the long-term reward at t is defined as $y_t = \sum_{k=t}^{\infty} \gamma^{k-t} r_k$, where $\gamma \in [0, 1)$ is a discounted factor. Note there is no expectation being taken here because there is no assumption of a probabilistic environment. Given a function space $\mathcal{F}$, the learner chooses a predictor f at each time step from $\mathcal{F}$ for predicting long-term rewards. In this section, we assume that any prediction made by a predictor f at a state $\mathbf{x}$ is upper bounded as $|f(\mathbf{x})| \leq P \in \mathbb{R}^+$, for any $f \in \mathcal{F}$ and $\mathbf{x}$.

At time step $t = 0$, the learner receives $\mathbf{x}_0$, initializes a predictor $f_0 \in \mathcal{F}$ and makes prediction $\hat{y}_0$ of y_0 as $f_0(\mathbf{x}_0)$. Rounds of learning then proceed as follows: the learner makes a prediction $\hat{y}_t$ of y_t at step t as $f_t(\mathbf{x}_t)$; the learner observes a reward r_t and the next state $\mathbf{x}_{t+1}$; the learner updates its predictor to f_{t+1}. This interaction repeats and is terminated after T steps.

We denote the goal of estimating the long-term discounted sequence of rewards in this setting as the problem of *online prediction of long-term reward*, PE.

Definitions

We first define the *signed Bellman Error* at step t for predictor f_t as $b_t = f_t(\mathbf{x}_t) - r_t - \gamma f_t(\mathbf{x}_{t+1})$, which measures effectively how self-consistent f_t is in its predictions between time step t and $t + 1$. We define the corresponding *Bellman Loss* at time step t with respect to predictor f as:

$$\ell_t^b(f) := (f(\mathbf{x}_t) - r_t - \gamma f(\mathbf{x}_{t+1}))^2. \tag{9.5.1}$$

The *Signed Prediction Error* of long-term reward at t for f_t is defined as $e_t = f_t(\mathbf{x}_t) - y_t$ and $e_t^* = f^*(\mathbf{x}_t) - y_t$ for f^* accordingly. What we actually care about is the long term *Prediction Error* (PE) e_t^2 of a given algorithm in terms of the best possible PE within our class of hypotheses. [7]

We can also define an online version of *TD Loss* at step t as:

$$\ell_t^d(f) := (f(\mathbf{x}_t) - r_t - \gamma f_t(\mathbf{x}_t))^2. \tag{9.5.2}$$

While we won't consider proving any results in this section, we note that more sophisticated versions of the arguments for Bellman error can be applied to the TD-error allowing us to develop a theory and new set of algorithms. The results are more difficult and more limited so we defer those to the literature [8]. [9]

Understanding Bootstrapping in Online Learning Setting

The true loss that a learner should care about is PE: $(f_t(\mathbf{x}_t) - y_t)^2$. However directly apply no-regret online algorithms on PE is not realistic in practice since in order to get y_t—the discounted sum of future rewards, one has to wait to get all rewards $\{r_i\}$ (or some truncation of this) for $t \leq i \leq T$. On the other hand, the algorithms we've discuss in this chapter use bootstrapping, which leverages the current predictor f_t to estimate y_t as $y_t \approx r_t + \gamma f_t(\mathbf{x}_{t+1})$.

This suggests a different perspective on temporal difference learning and residual gradient learning: *In the online learning setting, RG and TD both could be understood as running Online Gradient Descent on Bellman loss ℓ_t^b and TD loss ℓ_t^d, respectively.*

[7] To lighten notation in the following sections, all sums over time indices implicitly run from 0 to $T - 1$ unless explicitly noted otherwise.

[8] Wen Sun and J. Andrew (Drew) Bagnell. Online bellman residual and temporal difference algorithms with predictive error guarantees. In *Proceedings of The 25th International Joint Conference on Artificial Intelligence - IJCAI 2016*, April 2016

[9] As we noted earlier, though classic TD algorithm's update step is extremely similar to stochastic gradient descent, there is actually no well-defined objective function on which TD is performing stochastic gradient descent. The online view however provides a clear sequence of objectives and a clear goal of regret minimization.

At every time step t, after receiving the Bellman loss $\ell_t^b(f)$, let us consider what happens if apply online gradient descent on $\ell_t^b(f)$:

$$f_{t+1} = f_t - \mu_t b_t(\nabla_f f_t(\mathbf{x}_t) - \gamma \nabla_f f_t(\mathbf{x}_{t+1})), \tag{9.5.3}$$

where we denote $\nabla_f f(\mathbf{x})$ as the functional gradient of the evaluation functional $f(\mathbf{x})$ at function f.[10] Now for linear function approximation where $f(\mathbf{x})$ is represented as $\mathbf{w}^T\mathbf{x}$, the update step in Eq. 9.5.3 becomes:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \mu_t(\mathbf{w}_t^T\mathbf{x}_t - r_t - \gamma \mathbf{w}_t^T\mathbf{x}_{t+1})(\mathbf{x}_t - \gamma \mathbf{x}_{t+1}), \tag{9.5.4}$$

which reveals the RG algorithm proposed by [11]. [12]

Online Gradient Descent is one of the popular no-regret online learning algorithms. The above perspective suggests that RG and TD could be understood as applying a special no-regret online algorithm—OGD, to the Bellman loss and TD loss. This new perspective then naturally motivates the question: can any other no-regret online algorithms, such as Online Newton step, Online Frank Wolf and implicit online learning, be applied to Bellman loss ℓ_t^b and TD* loss $\ell_t^{d^*}$, and achieve guarantees on the long term loss we actually care about PE?

More formally, what one might hope is that if we can find a time-sequence of predictors that achieve the no-regret guarantee on TD loss $\{\ell_t^d\}$ or Bellman loss ℓ_t^b, then for the sequence of predictors $\{f_t\}$, the **real loss** we care about $\sum e_t^2$ could also be upper bounded via some competitive ratio:

$$\lim_{T\to\infty} \frac{1}{T}\sum e_t^2 \leq C \frac{1}{T}\sum e_t^{*2}, \quad \forall f^* \in \mathcal{F}, \tag{9.5.5}$$

where $C \in \mathbb{R}^+$ is constant. Schapire and Warmuth [1996] and later Li [2008] proved variants of this for particular gradient-style algorithms on these loss functions.

Bounding Long Term Predictive Regret

What we can show is that if an online algorithm running on the sequence of loss $\{l_t(f)\}$ is no-regret and the generated sequence of predictors $\{f_t\}$ satisfies a stability condition (we'll detail below), prediction error can indeed be upper bounded in the form of Eq. 9.5.5. The analysis is elementary, and use only a telescoping of the error terms combined with classical Cauchy-Schwartz bounds. What makes the analysis cool, besides the simplicity of the toolkit requires, is that it does **not** place any probabilistic assumption whatever on the sequence of observations $\{\mathbf{x}_t\}$ nor any assumption on the form of predictors $f \in \mathcal{F}$ (e.g., $f(\mathbf{x})$ does not have to be linear).

We start by first showing two key lemmas below:

Lemma 3. *Let us define $d_t = f_t(\mathbf{x}_t) - r_t - \gamma f_{t+1}(\mathbf{x}_{t+1})$. We have:*

$$\sum d_t^2 \geq (1-\gamma)^2 \sum e_t^2 + (\gamma^2 - \gamma)(e_T^2 - e_0^2). \tag{9.5.6}$$

Proof: At the heart of dynamic programming is a tele-scoping of terms. Schapire and Warmuth [1996] used such to a telescopng of term to implicitly show that $d_t = (f_t(\mathbf{x}) - v_t + v_t - (r + \gamma f_{t+1}(\mathbf{x}_{t+1}))) = (e_t - \gamma e_{t+1})$. Squaring

[10] We assume the function $\nabla_f f(\mathbf{x})$ belongs to $\mathcal{F}$. This is true for function classes such as Reproducing Kernel Hilbert Space (RKHS).

[11]

[12] Without the use of *double samples* to handle stochasticity.

both sides and summing over from $t = 0$ to $t = T - 1$, we get:

$$\begin{aligned}
&\sum d_t^2 = \sum (e_t - \gamma e_{t+1})^2 \\
&= \sum e_t^2 + \gamma^2 \sum e_{t+1}^2 - 2\gamma \sum e_t e_{t+1} \\
&\geq \sum e_t^2 + \gamma^2 \sum e_{t+1}^2 - \gamma \sum e_t^2 - \gamma \sum e_{t+1}^2 \\
&= (1-\gamma)^2 \sum e_t^2 + (\gamma^2 - \gamma)(e_T^2 - e_0^2). \qquad (9.5.7)
\end{aligned}$$

The first inequality is obtained by applying Young's inequality to $2e_t e_{t+1}$ to get $2e_t e_{t+1} \leq e_t^2 + e_{t+1}^2$. □ ■

In words, this tells us squared TD *lossupperboundsthelongtermpredictiveerror*(*moduloaboundaryterms*).

Lemma 4. *For any $f^* \in \mathcal{F}$, the prediction error $\sum e_t^{*2}$ upper bounds the BE $\sum b_t^{*2}$ as follows:*

$$\sum b_t^{*2} \leq (1+\gamma)^2 \sum e_t^{*2} + (\gamma + \gamma^2)(e_0^{*2} - e_T^{*2}). \qquad (9.5.8)$$

The proof of Lemma 4 is very similar to the one for Lemma. 3 and left as an exercise.

Now let us define a measure of the change in predictors between the steps of the online algorithm as $\epsilon_t = f_t(\mathbf{x}_{t+1}) - f_{t+1}(\mathbf{x}_{t+1})$, which is closely related to notions of online stability. We're going to require this change to be asymptotically controlled to get good performance. Then b_t and d_t are then closely related with each other by ϵ_t:

$$\begin{aligned}
d_t &= f_t(\mathbf{x}_t) - r_t - \gamma f_{t+1}(\mathbf{x}_{t+1}) - \gamma f_t(\mathbf{x}_{t+1}) + \gamma f_t(\mathbf{x}_{t+1}) \\
&= b_t + \gamma \epsilon_t.
\end{aligned}$$

Squaring both sides, we get:

$$\begin{aligned}
d_t^2 &= b_t^2 + 2b_t \gamma \epsilon_t + \gamma^2 \epsilon^2 \leq b_t^2 + b_t^2 + \gamma^2 \epsilon_t^2 + \gamma^2 \epsilon_t^2 \\
&= 2b_t^2 + 2\gamma^2 \epsilon_t^2, \qquad (9.5.9)
\end{aligned}$$

where the first inequality is coming from applying Young's inequality to $2b_t \gamma \epsilon_t$ to get $2b_t \gamma \epsilon_t \leq b_t^2 + \gamma^2 \epsilon_t^2$. In words, this tells us the signed temporal difference is exactly the signed Bellman error added to how much the our predictors disagree on x_{t+1}, and thus we can upper bound the TD *errorbythe* BE *andastabilityterm.Thistightlyconnectstheideaoftemporaldifferenceandbellmanerrors*.

We are now ready to state the following main theorem of this paper:

Theorem 5. *Assume a sequence of predictors $\{f_t\}$ is generated by running some online algorithm on the sequence of loss functions $\{l_t\}$. For any predictor $f^* \in \mathcal{F}$, the sum of prediction errors $\sum e_t^2$ can be upper bounded as:*

$$\begin{aligned}
(1-\gamma)^2 \sum e_t^2 \leq & 2 \sum (b_t^2 - b_t^{*2}) + 2\gamma^2 \sum \epsilon_t^2 \\
& + 2(1+\gamma)^2 \sum e_t^{*2} + M, \qquad (9.5.10)
\end{aligned}$$

where

$$M = 2(\gamma + \gamma^2)(e_0^{*2} - e_T^{*2}) - (\gamma^2 - \gamma)(e_T^2 - e_0^2).$$

By running a no-regret and online stable algorithm on the loss functions $\{l_t(f)\}$, as $T \to \infty$, the average prediction error is then asymptotically upper bounded by a constant factor of the best possible prediction error in the function class:

$$\lim_{T \to \infty} : \frac{\sum e_t^2}{T} \leq \frac{2(1+\gamma)^2}{(1-\gamma)^2} \frac{\sum e_t^{*2}}{T}. \qquad (9.5.11)$$

Proof: Combining Lemma. 3 and Lemma. 4, we have:

$$\begin{aligned}
&\sum d_t^2 - 2\sum b_t^{*2} \\
&\geq (1-\gamma)^2 \sum e_t^2 + (\gamma^2-\gamma)(e_T^2 - e_0^2) \\
&- 2(1+\gamma)^2 \sum e_t^{*2} \\
&- 2(\gamma+\gamma^2)(e_0^{*2} - e_T^{*2}).
\end{aligned} \tag{9.5.12}$$

Subtracting $2b_t^{*2}$ on both sides of Eq. 9.5.9, and then summing over from $t = 1$ to $T-1$, we have:

$$\sum d_t^2 - \sum 2b_t^{*2} \leq 2\sum (b_t^2 - b_t^{*2}) + 2\gamma^2 \sum \epsilon_t^2.$$

Combining the above two inequalities together, we have:

$$\begin{aligned}
&2\sum (b_t^2 - b_t^{*2}) + 2\gamma^2 \sum \epsilon_t^2 \\
&\geq (1-\gamma)^2 \sum e_t^2 + (\gamma^2-\gamma)(e_T^2 - e_0^2) \\
&- 2(1+\gamma)^2 \sum e_t^{*2} - 2(\gamma+\gamma^2)(e_0^{*2} - e_T^{*2}).
\end{aligned} \tag{9.5.13}$$

Rearrange inequality (9.5.13) and define $M = 2(\gamma+\gamma^2)(e_0^{*2} - e_T^{*2}) - (\gamma^2 - \gamma)(e_T^2 - e_0^2)$, we obtain inequality (9.5.10)

Assume that the $\bar{f} = \arg\min_{f\in\mathcal{F}} \sum l_t(f)$, then if the online algorithm is no-regret, we have

$$\begin{aligned}
&\frac{1}{T}\sum b_t^2 - b_t^{*2} = \frac{1}{T}\sum l_t(f_t) - l_t(f^*) \\
&\leq \frac{1}{T}\sum l_t(f_t) - l_t(\bar{f}) \\
&= \frac{1}{T}\,\text{Regret} \leq 0, \quad T \to \infty.
\end{aligned} \tag{9.5.14}$$

If, further, our choice of online algorithm satisfies a *stability condition*, we can remove a term. For the generated sequence of predictors f_t, we say the algorithm is online stable if:

$$\lim_{T\to\infty} \frac{1}{T}\sum (f_t(\mathbf{x}_{t+1}) - f_{t+1}(\mathbf{x}_{t+1}))^2 = 0. \tag{9.5.15}$$

Intuitively, this form online stability means that on average the difference between successive predictors is eventually small on average. Online stability is a general condition and does not notably limit the scope of the online learning algorithms. [13]

Thus, assuming stability, we have: $\frac{1}{T}\sum \epsilon_t^2 = 0$ when $T \to \infty$.

Also, since we assume $|f(\mathbf{x})| \leq P$ and $|r| \leq R$, we can see M must be upper bounded by some constant. Hence, we must have $\frac{M}{T} = 0$, as $T \to \infty$.

Under the conditions that the online algorithm is no-regret and satisfies online stability, we get Eq. 9.5.11 by dividing both sides of Eq. 9.5.10 by T and taking T to infinity. □ ■

Note that in the Thm. 5, Eq. 9.5.10 holds for any $f^* \in \mathcal{F}$, including the f^* that minimizes the prediction error.

Conclusion

It is interesting that we can derive and prove bounds for a wide class of algorithms without making the markov assumption. A natural question is

[13] For instance, when f is linear, the definition of stability of online learning in (see Eq. 3 in) and imply this kind of stability. In fact, almost all popular no-regret online learning algorithms satisfy this condition. Moreover, it turns out that this stability is actually essential– without this requirement we can generate counterexamples to the having any competitive ratio at all.

Ankan Saha, Prateek Jain, and Ambuj Tewari. The Interplay Between Stability and Regret in Online Learning. *arXiv preprint arXiv:1211.6158*, pages 1–19, 2012. URL http://arxiv.org/abs/1211.6158; Ankan Saha, Prateek Jain, and Ambuj Tewari. The Interplay Between Stability and Regret in Online Learning. *arXiv preprint arXiv:1211.6158*, pages 1–19, 2012. URL http://arxiv.org/abs/1211.6158; and Stephane Ross and J. Andrew Bagnell. Stability Conditions for Online Learnability. *arXiv:1108.3154*, 2011. URL http://arxiv.org/abs/1108.3154

whether we can use these approaches to design algorithms that are more robust to distributional shift, or to understand the superior performance of online methods like TD as compared with offline methods like fitted value-iteration.

10
Black-Box Policy Optimization

Up to this point we have learned primarily about dynamic programming-based approaches to control. These problems set up a Bellman equation which can be solved to discover an optimal or approximately optimal controller. This lecture focuses on ways to avoid relying on Bellman equations and the perils of dynamic programming and distribution shift. Put more poetically by Andrew Moore, this chapter focuses on how not to be "blinded by the beauty of the Bellman equation."

The following approaches will focus on finding a set of parameters which defines a good controller. For example, in Tetris, we could imagine defining a policy $\pi_\theta : x \mapsto a$ which is parameterized by θ. These parameters might be weights on various features defined on state-action pair (x, a), such as the maximum height of the board or the number of holes of the resulting configuration. A policy under this parameterization can be defined at every state x as,

$$\pi_\theta(x) = \underset{a \in \mathbb{A}}{\operatorname{argmin}} \left(\theta_1 \times \#\text{ of Holes}(x, a) + \theta_2 \times \text{Height}(x, a)\right).$$

In general, we have

$$\pi_\theta(x) = \underset{a \in \mathbb{A}}{\operatorname{argmin}} \, \theta^T f(x, a),$$

where $f(x, a)$ is a vector of features of the state-action pair (x, a).

Let ξ denote a *trajectory* of states and actions, $\xi = (x_0, a_0, \ldots, x_{T-1}, a_{T-1})$. We define the *total reward of the trajectory* ξ as,

$$R(\xi) = \sum_{t=0}^{T-1} r(x_t, a_t).$$

Our goal is to find the parameters that produce the policy that maximizes the expected total reward of the trajectories,

$$J(\theta) = E_{p(\xi|\theta)}[R(\xi)] = E_{p(\xi|\theta)}\left[\sum_{t=0}^{T-1} r(x_t, a_t)\right],$$

where $p(\xi|\theta)$ is the probability of the trajectory ξ given the policy parameterized by θ.

There are tremendous advantages to this simple, stochastic optimization viewpoint on the reinforcement learning problem:

Pros of Policy Optimization with Parameterized Policies:

- **No** dependence on size of state space (directly)
- A policy can be much simpler than a value function. For example, in the famed mountain car problem [1], the optimal policy is simple to specify: move backwards until a certain point, then move forwards. The value function for this problem, however, is significantly more complex.
- Engineering knowledge about the domain can be put directly into the policy by selecting good features.
- Only needs a crude reset access model to optimize the policy directly.
- Simple and easy to code up!

[1] https://en.wikipedia.org/wiki/Mountain_car_problem

Cons of Policy Optimization with Parametrized Policies:

- Needs careful design of features. With poor features, no amount of searching will find a good policy. Also, the features need to have somewhat smooth gradients for this type of gradient descent to be effective.
- Strong dependence on the number of parameters. Irrelevant or redundant parameters make the problem much harder (potentially exponentially harder).
- Exploration is particularly difficult in this setting and can lead to exponentially slow convergence in the number of states.
- As we'll discuss in the next chapter, we're ignoring key, known, information including the relationship (e.g. Jacobian) between parameters and action choices and the markov structure of states and rewards. This may come at significant sample complexity cost.

10.1 How to find a good parameter set θ?

Gradient Ascent/Descent

Perhaps the most obvious way to solve this problem would be to use a gradient ascent style algorithm. Gradient ascent starts at some intial point, evaluates the gradient of the objective function $J(\theta)$, which is the expected total reward function in our case, and then takes a step up the gradient (if we are maximizing). Gradient ascent continues stepping in the direction of the gradient (the direction that the function J has the greatest rate of increase) until it converges, i.e., the gradient is small enough.

Problem 1:

- $J(\theta)$ may not be differentiable, i.e., changing θ by a infinitesimal small change δ could cause J to jump substantially
- $J(\theta)$ may be very hard to differentiate analytically, particularly because, by assumption we only have *oth* order access. That is, we can draw samples, but we can't get access to derivates of the world dynamics.

Idea: approximate the gradient using finite differences

For each parameter we could add a small scalar δ to it and evaluate the value of J at $\theta + \delta_i$, where $\delta_i = (0, \ldots, 0, \delta, 0, \ldots, 0)$.

Then, we can use the finite difference $\frac{1}{\delta}(J(\theta + \delta_i) - J(\theta))$ to estimate the derivative in the ith direction. The estimated gradient then is

$$\widetilde{\nabla} = \frac{1}{\delta} \cdot \begin{bmatrix} J(\theta + \delta_1) - J(\theta) \\ \vdots \\ J(\theta + \delta_n) - J(\theta) \end{bmatrix}.$$

Problem 2:

- We may not have access to the value of $J(\theta)$, rather, we may have a noisy sample $\tilde{J}(\theta)$, which is the case for the expected total reward function.

Idea: estimate the gradient using the samples.

Similarly, we could add a small scalar δ to each parameter and take a single *sample* $\tilde{J}(\theta + \delta_i)$ to estimate the derivative in the ith direction. The estimated gradient is

$$\widetilde{\nabla} = \frac{1}{\delta} \cdot \begin{bmatrix} \tilde{J}(\theta + \delta_1) - \tilde{J}(\theta) \\ \vdots \\ \tilde{J}(\theta + \delta_n) - \tilde{J}(\theta) \end{bmatrix}.$$

However, this estimate can be noisy. If we want a better estimate of the gradient, we could sample multiple times and take an average. A better way would be to use a linear least squares approach for a large number of sample vectors. Specifically, we create tuples, $\{\Delta^{(j)}, \tilde{J}(\theta + \Delta^{(j)}) - \tilde{J}(\theta)\}_{j=1}^{N}$. Then, by the Taylor series expansion, we have,

$$\tilde{J}(\theta + \Delta^{(j)}) - \tilde{J}(\theta) \approx (\nabla_\theta J)^\top \Delta^{(j)}$$

Then, the problem of estimating gradient can be interpreted as the following linear least squares regression problem,

$$\tilde{\nabla} = \underset{\nabla'}{\operatorname{argmin}} \sum_{j=1}^{N} \left| (\nabla')^\top \Delta^{(j)} - \left(\tilde{J}(\theta + \Delta^{(j)}) - \tilde{J}(\theta) \right) \right|^2.$$

In the next lecture, we will see other methods to estimate $\nabla_\theta J$ called the *policy gradient methods*.

In some domains, such as a deterministic simulator (although the simulator may simulate randomness, it itself is deterministic, such as Tetris), we can use the so called *Pegasus [1] trick*: simply fix the random seed. This can be useful because it fixes a single (noisy) estimate of the true gradient and helps keep the gradient consistent. This can be dangerous because it is sacrificing bias to obtain a lower variance estimate and may drive θ towards areas areas that are not actually a local optima.

With the gradient estimate, we can update the parameter θ:

$$\theta \leftarrow \theta + \alpha \widetilde{\nabla}$$

where α is the *step size* or *learning rate*. In practice, for good convergence we need $\alpha \approx \frac{1}{\sqrt{T}}$ where T is the time horizon of the problem.

Note, however, that poor gradient estimates can cause incorrect behavior. In the worst case, the estimated gradient near an almost flat section could be 0 in all directions.

Alternative and Useful oth-Order Optimization Algorithms

In addition to the algorithms covered in more detail below, it may be worth considering other black box techniques:

- Nelder-Mead. At least one of those others always had good luck with this method (sometimes called the simplex method). [2]
- CMA-ES and Cross-Entropy [3] These algorithms balance explortation with "gradient-like" exploitation by maintaining a probability distribution over parametric hypthothesis. We detail one variant below.
- Simulated annealing. This method performs gradient descent like updates (more precisely, hill-climing updates). At each iteration, another set of parameters $\theta + \Delta$ is randomly generated with a small Δ, if $J(\theta + \Delta) > J(\theta)$, we update the parameters $\theta \leftarrow \theta + \Delta$. Otherwise, we still accept the update $\theta \leftarrow \theta + \Delta$ with some probability related to the "temperature" of the system. Initially, the "temperature" is high which means the algorithm tends towards random movement, i.e., even if the value is not better, we still make the updates with high probability. As the search continues the temperature decreases and the algorithm is more likely to move in the ascent direction.
- Genetic Algorithms. These are generally a method of last resort. They evaluate a bunch of random parameters and then the best parameters "survive" and "reproduce" with some "mutation" to create a new set of parameters. This method is nice because it requires basically no knowledge of the problem and, when tuned properly, will explore the space nicely, although it can be very difficult to tune the hyper-parameters of these approaches.
- Q2. This method generates a bunch of samples and fits a quadratic, then solves a quadratic program to optimize the weights. To avoid running outside the region about which the algorithm "quadraticized", it applies linear constraints to bound the solution. It then re-quadraticizes about the new estimate. [4]
- Coordinate Descent. In order to find a minimum, this algorithm performs a line search along one coordinate direction at the current point during each iteration. Different coordinate directions are cycled through as the algorithm iterates.

[2] ; ; and

[3]

[4]

Nelder-Mead

(See the wikipedia article: http://en.wikipedia.org/wiki/Nelder%E2%80%93Mead_method for more info and a nice animated gif)

The Nelder–Mead method was proposed by John Nelder and Roger Mead, two English statisticians working at the National Vegetable Research Station[5]. Perhaps the best summary for the Nelder–Mead method is what Nelder said himself during an interview [3]:

> "There are occasions where it has been spectacularly good... Mathematicians hate it because you can't prove convergence; engineers seem to love it because it often works."

Nelder-Mead has many popular variants, one of which is the default algorithm used in MATLAB's `fminsearch` function. It does not require any

[5] Nelder later notes that *"Our address (National Vegetable Research Station) also caused surprise in one famous US laboratory, whose staff clearly doubted if turnipbashers could be numerate."* [3]

knowledge of the derivatives or the analytic form of the function being optimized, but it does expect deterministic functions.

Nelder-Mead works on an n-dimensional function by creating a simplex of $n + 1$ points which it modifies to try to surround the optimum. At each iteration, it evaluates the function at each of the vertices of the simplex and follows some complicated rules to move the points until it shrinks the simplex down on a local minima. The original version of the algorithm is not guaranteed to converge.

The following is an overview of the rules used:

- Consider points along the line between the worst point and the (possibly weighted) average of the other points
- Try to reflect the worst point about plane between other points
  - If the reflected point is better than the second worst, but not better than the best, replace the worst with the reflected point.
  - If the reflected point is better than best point, compute a further expanded point past the reflected point. If this point is better than the reflection, replace the worst point with it, otherwise replace the worst point with the reflection.
  - If neither are better, consider contracting the simplex by shortening the distances between the best point and the other points

Note: you should really consult [6] and other references if you were considering implementing this in anger as there are many variants of this algorithm.

[6]

Even though it may not have good theoretical properties, in practice this algorithm tends to be very effective. This approach can also be extended to take 4 or 8 samples at each point on the simplex instead of just sampling once. These methods (Nelder-Mead-4 / Nelder-Mead-8) can potentially improve robustness to noise.

Cross-Entropy Method

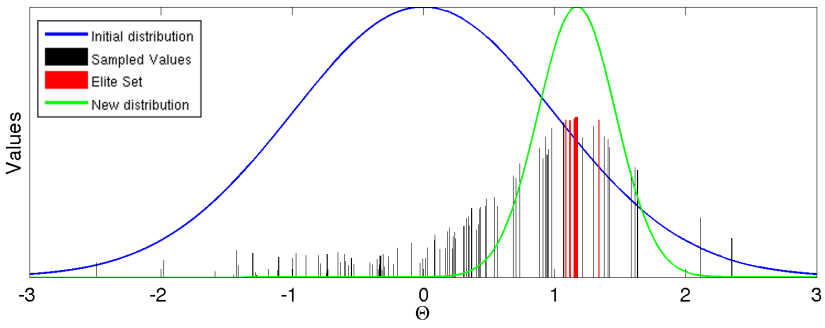


Figure 10.1.1: The first iteration of cross-entropy. The initial distribution is a prior Gaussian (blue) and the green Gaussian is the one fitted to the elite set.

The Cross-Entropy Method [7] [8] samples from a distribution, and then updates the distribution based on which samples scored the highest. This method originated as an approach for importance sampling and has impacts

[7]

[8] See in particular Algorithm 4.1 of the reference.

in queuing theory as well as being useful as an optimization technique. It's a surprisingly effective brute force method.

The method, shown in Algorithm 19 and illustrated in Figure 10.1.1 starts with a distribution over the parameter space, often a Gaussian, but it can be any distribution. Then samples are taken from the distribution as points at which to evaluate the function. Typically about 100 samples are taken. Then the "elite set" is computed, which is the top $1-5\%$ of the samples. The parameters that make up the elite set are then used to create a new distribution. The actual values of the elite set are ignored, only their parameters are used to train a new distribution. Then the new distribution is sampled from and the process repeats until the distribution settles in on a local optimum. The parameters returned could be the mean of the final distribution, or one could track the best value overall and use that as the final parameter set.

Algorithm 19: Cross entropy method

given: An initial distribution $\mathcal{D}_\theta$ over the set of parameters
outputs: A final set of parameters θ_n
while not converged **do**
  for $i = 1$ to k **do**
    sample θ_i from $\mathcal{D}_\theta$
    $v_i \leftarrow J(\theta_i)$ {Run the simulator to obtain a value}
  end for
  $E \leftarrow \emptyset$
  for $j = 1$ to e **do**
    $i \leftarrow \text{argmax}_{i \notin E}\, v_i$
    $E \leftarrow E \cup \theta_i$ {Find the e best values to create the elite set}
  end for
  $\mathcal{D}_\theta \leftarrow \text{fit}(E)$ {Fit a new distribution to the x_j in the elite set}
end while

This method has an interesting set of guarantees, as it has nice exploration property since it samples randomly at each step. [9]

[9]

One possible modification is to mix the old and new distributions, such as by linearly interpolating the mean and covariance in the case of Gaussians and in general regularizing the learned distribution. Typically the interpolation is weighted $70-90\%$ in favor of the new distribution. This modification is useful to help avoid singular covariance matrices.

Another nice property of Cross-Entropy is that it can deal with irrelevant or noisy features. If two features are related, their covariance in the distribution will be high.

There are, however, issues with these methods

- Inaccuracies in modeling the true distribution. If the actual distribution is multi modal, then that can cause the covariance to keep growing to accommodate the bimodal nature of the underlying distribution
- If the sampling is not done right, then there might be too few elements in the covariance matrix. To fix this, some people try to increase the diagonals along the covariance by adding a 'regularizer' term to the covariance

matrix, i.e. a λI, or by linearly combining the distributions as mentioned earlier.

- This method actually optimizes quantiles [2] rather than the actual expected values. Thus, if using a black box method, it will converge, but if a stochastic policy method is used, it will not converge because of noise.

Black box methods usually must evaluate J many times, and thus work well when evaluating J is cheap. However, this is almost never the case in robotics. Their simplicity and robustness to incorrect modeling assumptions (partial observability and difficult approximating value functions) make them particularly appealing and often they can be used on a learned model of a system to improve sampl efficiency. We study the use of learned models in a later lecture.

10.2 Related Reading

[1] PEGASUS: A policy search method for large MDPs and POMDPs. Ng, Andrew Y and Jordan, Michael

[2] The Cross-Entropy Method Optimizes for Quantiles. Goschin, Weinstein and Littman

[3] Optimization stories. GrÃűtschel, Martin, ed. Dt. Mathematiker-Vereinigung, 2012.

11
Policy Gradients

In this lecture, we will continue to consider the problem of directly learning a policy including from sampled trajectories. We will focus on *policy gradient methods* that use samples from the environment to get noisy gradient estimates and then update policy. Policy gradient methods take advantage of one important structure black box methods do not: the fact that we can design our policy space such that we know the relationship between the parameters of that policy space and the output actions. That is, the policy search problem need not be *entirely* a black-box operation since even without a model of the environment or cost functions as we still can have a model of how our policy space works. [1]

To take advantage of this approach, we need a method to relate the parameters of the policy class with the resulting actions. One of the simplest methods to do so is computing derivatives: for sophisticated policy classes this is often best done through automatic differentiation techniques. We begin here by reviewing the most common automatic differentiation technique used in the learning literature commonly known as backpropagation or *reverse-mode automatic differentiation*.

We demonstrate how to use this in a larger loop of policy optimization later in the lecture. Before specifying the details of this approach, we will review back-propagation and its use in neural networks and controls.

11.1 Back-propagation

A powerful way to describe many complex systems is as a composition of interconnected modules described as a *directed graph*[2]. This makes it easier to organize a complex system and debug the system by unit testing individual components. For example, robotics often leverages a "sense, act, plan" paradigm, where each component is often studied and optimized separately. However, modifying a single module can influence the overall system performance in a complicated way due to the relationship between modules. Back-propagation attempts to address this problem by offering a principled method to calculate the cascaded effects of module parameters on overall system performance. Back-propagation is also known as *the adjoint method* and *reverse-mode automatic differentiation* in the control and optimization literature.

Back-propagation makes it possible to solve a large class of problems that would be intractable using naive differentiation techniques or the use of

[1] At least one of the authors, based on disappointing experimental evidence, had largely despaired of this advantage translating into a reduction in the amount of interaction required over naive blackbox methods "A major open issue within the field is the relative merits of the these two approaches: in principle, white box methods leverage more information, but with the exception of models, the performance gains are traded-off with additional assumptions that may be violated and less mature optimization algorithms. Some recent work ... suggest that much of the benefit of policy search is achieved by black-box methods." (Kober, Peters, Bagnell, 14). In recent years, as policy classes have become more sophisticated (i.e. deep CNN based policies) with very large parameter sets, the benefit of this additional structure has become important and the methods described in this chapter have at times become preferred to black box search.

[2] Often called the *computation graph* in the learning literature

forward mode auto-differentiation. One of the best known applications is training neural networks, which has led to dramatic results in computer vision and natural language processing. [3] A common misunderstanding is that back-propagation is specific to machine learning/neural network (*aka deep network*) training. In fact, back-propagation can be used to compute gradients for any differentiable function expressed as a graph of operations and this general dynamic programming strategy for computing derivatives is quite old. In particular, the same idea has been used widely in optimal control, known as *the adjoint method*[4]. Just as back-propagation's led to tremendous success in training neural nets, the adjoint method has enabled researchers to tackle complex control problems with millions of control inputs. An elegant example is the work "Fluid Control with the Adjoint Method" [1], where the simulation of a human-shaped smoke cloud required over one million control inputs. Naive derivative computation with this number of parameters is nearly intractable, while backpropagation allows it to scale to real time animation. As the backpropagation technique has become better understood and more ubiquitous, we've entered a period of *differentiable programming* [5] where we can assemble sophisticated programs and their derivatives to enable optimization of these programs.

[3]

[4] A fine summary of the adjoint method can be found here: http://www.argmin.net/2016/05/18/mates-of-costate/

[5] https://en.wikipedia.org/wiki/Differentiable_programming

We will first look at back-propagation as a general algorithm to compute gradients, then we will see several examples including multi-layer neural networks and the LQR problem. An excellent reference on the origins and general backpropagation technique is [6]. The book *Deep Learning* [7] provides a fine introduction in section 6.5.

[6]

[7]

Total vs. Partial Derivatives

In dealing with compositions of functions, a crucial distinction must be made between two types of derivatives, **total derivatives** and **partial derivatives**. The partial derivative of a function describes the change in output resulting from a change in direct dependencies– i.e. a module has a set of inputs and we evaluate how the output of the module changes in terms of these inputs. The total derivative of a function describes the change of the output resulting from *all* dependencies, direct and indirect. For instance, in a control problem, a module describing the result of dynamics at time time t, x_{t+1} may have no direct dependence on a control u_{t-5} at time step $t-5$ and hence has *partial derivative* of 0. However, there is a potentially non-0 *total derivative* of that output in terms of u_{t-5}, as this control effects the output, albeit indirectly.

In a sense, *partial derivatives* are "syntactic" and *total derivates* are semantic, representing the complete effect of varying a single parameter or input on a resulting computation.

The Chain Rule

In other terms, the partial derivative does not account for the composition but rather direct inputs, while the total derivative does. Backpropagation is effectively a dynamic programming means to turn manually specified partial derivatives into automatic computation of total derivatives.

Before diving in and solving more sophisticated problems using back-propagation, let's review some basic calculus starting with the chain rule of calculus. First, let us consider the simplest case where $x \in \mathbb{R}$ is a real num-

ber. Let f and g be two differentiable functions that map $\mathbb{R}$ to $\mathbb{R}$. Suppose that $y = g(x)$ and $z = f(y) = f(g(x))$. Then, the chain rule tells us,

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{11.1.1}$$

The chain rule can further generalized to the case when $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$ are vectors[8]. Let $f : \mathbb{R}^m \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^m$ be two differentiable functions. As before, suppose that $z = f(g(x))$. Then, we have,

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}. \tag{11.1.2}$$

In vector notation, we rewrite the above equation as,

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^\top \nabla_y z, \tag{11.1.3}$$

where $\nabla_x z = [\frac{\partial z}{\partial x_1}, \dots \frac{\partial z}{\partial x_n}]^\top$ and $\nabla_y z = [\frac{\partial z}{\partial y_1}, \dots \frac{\partial z}{\partial y_m}]^\top$ are the *gradient* of z with respect to x and y, respectively, and

$$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \dots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \dots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \dots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

is the *Jacobian matrix* of the function g.

[8] In fact, the chain rule can be generalized to the case of "tensors". The use of this phrase in *deep learning* doesn't imply the geometric meaning of mathematics, but rather simply refers to a multi-dimensional array of numbers. See

Block Diagrams

Now that we are equipped with the necessary mathematical tools to compute gradients, let us take one step further and look at how we can represent how the modules are interconnected in a system using a *block diagram*.

In the language of block diagram, each *module* or *operation* is represented by a block, whereas the arrows between blocks indicate variables that are inputs to/outputs of the operations. For example, the system considered in the previous section can be represented as Figure 11.1.1.

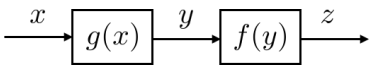


Figure 11.1.1: The block diagram representation of the simple example.

Given a block diagram and a variable x in the diagram, we say a variable y is a *parent*[9] of x if there exists a block f such that x is the output of f and y one of the inputs. Note that a variable may have multiple parents since there can be multiple inputs to block f. We denote the set of variables that are parents of x as $Parents(x)$. Conversely, we call a variable y as a *child* of x if x is a parent of y, i.e., there exists a block g such that y is the output of g and x is one of the inputs. We denote the set of variables that are children of x as $Children(x)$.

[9] Here we abuse the definition of parents by denoting an "edge" as a parent of another "edge" in the diagram. Same for children.

We say a block diagram is *acyclic* if it has no cyclic paths. For back-propagation, we assume that the associated block diagram is acyclic[10], and there exists a topological ordering (over variables) such that the output of the system is the last one in the list. In our case, we assume that the output of

[10] Recurrent neural networks and closed loop control systems (with a finite horizon) can be represented by an acyclic diagram through an operation called *unfold*. We will discuss it later.

the system is a scalar $J \in \mathbb{R}$. It could be the value of the loss function if we are training a neural network, it can also be the total cost of the trajectory(ies) if we are optimizing a policy.

Recall that we are interested how the output J is changed when we change a variable x in the diagram, which is precisely the gradient $\nabla_x J$. By the chain rule, we have,

$$\nabla_x J = \sum_{y \in Children(x)} \left(\frac{\partial y}{\partial x}\right)^\top \nabla_y J \tag{11.1.4}$$

Examples

To make things more concrete, let us look at some examples.

- *Linear*

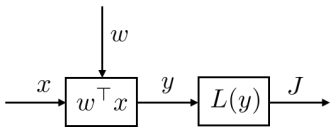


Figure 11.1.2: The block diagram of the linear module.

A *linear module* takes two inputs x and w to produce output $y = f(x, w) = w^\top x$. Assume that the system is associated with an overall output $J = L(y)$. Then, we have,

$$\left(\frac{\partial y}{\partial x}\right)^\top = w, \qquad \left(\frac{\partial y}{\partial w}\right)^\top = x \tag{11.1.5}$$

$$\nabla_x J = \left(\frac{\partial y}{\partial x}\right)^\top \nabla_y J = \frac{dL}{dy} w \tag{11.1.6}$$

$$\nabla_w J = \left(\frac{\partial y}{\partial w}\right)^\top \nabla_y J = \frac{dL}{dy} x \tag{11.1.7}$$

- *Squared Loss*

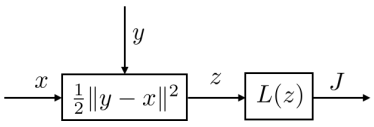


Figure 11.1.3: The block diagram of the squared loss module.

A *squared loss module* takes two inputs x and y and produces output $z = f(x, y) = \frac{1}{2}(y - x)^\top (y - x) = \frac{1}{2}\|y - x\|^2$. Assume that the system is associated with an overall output $J = L(z)$. Then, we have,

$$\left(\frac{\partial z}{\partial x}\right)^\top = x - y, \qquad \left(\frac{\partial z}{\partial y}\right)^\top = y - x \tag{11.1.8}$$

$$\nabla_x J = \left(\frac{\partial z}{\partial x}\right)^\top \nabla_z J = \frac{dL}{dz}(x - y) \tag{11.1.9}$$

$$\nabla_y J = \left(\frac{\partial z}{\partial y}\right)^\top \nabla_z J = \frac{dL}{dz}(y - x) \tag{11.1.10}$$

- *Branch*

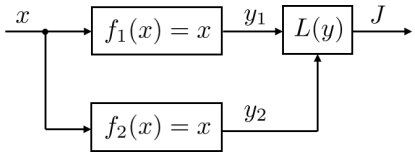


Figure 11.1.4: The block diagram of the branch module.

A *branch module* takes in one input x and produces two outputs $y_1 = f_1(x) = x$ and $y_2 = f_2(x) = x$. Assume that the system is associated with an overall output $J = L(y_1, y_2)$. Then, we have,

$$\left(\frac{\partial y_1}{\partial x}\right)^\top = \left(\frac{\partial y_2}{\partial x}\right)^\top = I \tag{11.1.11}$$

$$\nabla_x J = \left(\frac{\partial y_1}{\partial x}\right)^\top \nabla_{y_1} J + \left(\frac{\partial y_2}{\partial y}\right)^\top \nabla_{y_2} J = \nabla_{y_1} J + \nabla_{y_2} J \tag{11.1.12}$$

- *Addition*

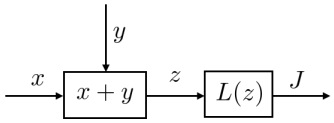


Figure 11.1.5: The block diagram of the plus module.

An *addition module* takes in two inputs x and y and produces output $z = f(x, y) = x + y$. Again, assume that the system is associated with an overall output $J = L(z)$. Then, we have,

$$\left(\frac{\partial z}{\partial x}\right)^\top = \left(\frac{\partial z}{\partial y}\right)^\top = I \tag{11.1.13}$$

$$\nabla_x J = \left(\frac{\partial z}{\partial x}\right)^\top \nabla_z J = \nabla_z J \tag{11.1.14}$$

$$\nabla_y J = \left(\frac{\partial z}{\partial y}\right)^\top \nabla_z J = \nabla_z J \tag{11.1.15}$$

Back-propagation: A Dynamic Programming Algorithm

Although given any variable x in the diagram, we can calculate the gradient of the output J with respect to the variable x by recursively applying the chain rule. However, when we are training a neural network or solving an optimal control problem, we oftentimes want to compute the gradient with respect to *a large set of variables*, such as weights in every layer of the neural network, or the control input at every time step. The question then becomes, can we do something better than calculating the gradients one by one? The answer is yes!

To see this, let us look back at the linear module example. When we calculate the $\nabla_x J$ and $\nabla_w J$ in (11.1.6) and (11.1.7), we actually use the value of $\nabla_y J$ for multiple times. Therefore, if we can somehow *store* the previously calculated gradients, and order the variables in such a way that we can make use of the gradients computed previously, then we can save a lot of computation by *reusing* these gradients. This idea of dynamic programming is the main idea behind *back-propagation*.

Recall from the previous part that the gradient with respect to a variable x can be computed based on the gradient with respect to all its children $y \in Children(x)$,

$$\nabla_x J = \sum_{y \in Children(x)} \left(\frac{\partial y}{\partial x}\right)^\top \nabla_y J.$$

Based on this observation, we see that in order to reuse the previously computed gradient, we need to order the variables *backwards* – from the output to the inputs, from the parents to the children. Then, we need to backward *propagate* the gradients from the children to the parents, this is where the name *back-propagation* comes from. [11]

[11] This dynamic programming ordering might remind you a bit of *value iteration* from the earliest lectures. It should! If you think of the output of the final module as the value function, backpropagation is simply doing value iteration with a local, linear approximation of the value function. As such, it's essentially value-iteration in disguise.

The "Learning" Algorithm

Now, let us try to do something useful with the back-propagation algorithm. Assume that there are a set of input variables in the diagram called *parameters* that we are free to choose. We denote these parameters as $\{w_i\}_i$. Examples of these parameters include weights in the neural networks, control inputs and initial conditions, etc. Conversely, there are other input variables whose values are given and we have no control over, such as the inputs to the neural network, the system dynamics, etc. Our goal is to find a set of *parameters* such that the value of some scalar output J is minimized (loss for training neural networks, cost for optimal control problems, etc), i.e.,

$$\{w_i^*\}_i = \arg\min_{\{w_i\}_i} J \tag{11.1.16}$$

We are interested in designing a learning algorithm that updates parameters of a system to reduce the value of J. One way to perform the gradient descent algorithm,

$$w_i^{k+1} = w_i^k - \alpha \nabla_{w_i} J. \tag{11.1.17}$$

where $\alpha > 0$ is the learning rate. Note that here the gradients can be calculated by the back-propagation algorithm.

In summary, there are three main steps in the learning algorithm: *forward-propagation*, *back-propagation*, and *gradient descent*. Forward propagation consists of generating all module outputs by running the system "forward" (from the inputs to the output). This is necessary for recursively evaluating all the partial derivatives in the back propagation step, as detailed in the previous section. Finally, once all gradients have been calculated, we take a gradient descent step. Then we repeat the whole process until convergence. [12]

[12] More sophisticated algorithms than gradient descent are at times used that automatically scale individual directions or apply approximations to a second order method. See for more details.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org

11.2 System Examples

Below are examples of modular systems where back-propagation can be used.

Linear Regression Example

We can describe a linear regression by a linear module cascaded with a squared loss module, as shown below. Linear module takes two inputs x

and w, and output $z = w^\top x$. Squared loss module takes inputs z and y, and output $J = \frac{1}{2}(z-y)^\top(z-y)$. The system takes inputs x, y, and w, where x corresponds to the data, y are the respective regression targets, and w is the regression parameter we can control. Our goal is to minimize the loss z. Back-propagation is usually not used here because it is not difficult to calculate the total derivatives directly.

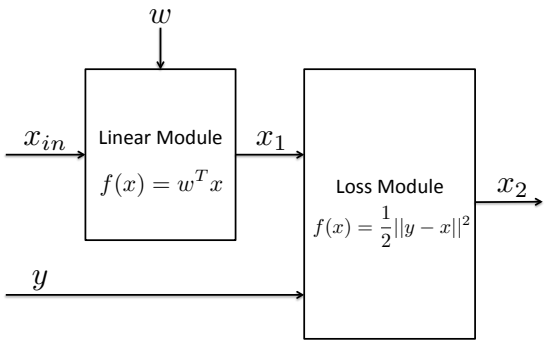


Figure 11.2.1: Linear regression represented as a cascade of modules.

Neural Networks

A neural network consists of layered linear modules and nonlinear firing units. Traditionally, the firing units are sigmoid functions such as hyperbolic tangent or the logistic function. Recently, the nonlinear rectifier function shown below has come into common practice. The sigmoid functions have small linear support regions between saturation, which require the inputs to be scaled properly. The rectifier does not suffer from these issues, and is computationally simpler, allowing for large neural networks to be applied to a variety of data.

In deep, multi-layer networks, cascading makes it difficult to directly determine total derivatives for all the parameters. Utilizing the back-propagation algorithm, however, we can efficiently tune the linear module weight parameters.

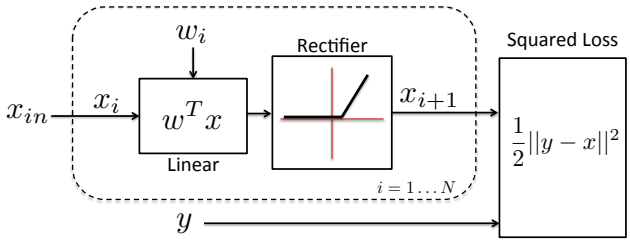


Figure 11.2.2: A neural net.

Let J be the output of the squared loss function. Then, we have,

$$\nabla_{x_{N+1}} J = x_{N+1} - y. \tag{11.2.1}$$

By the chain rule, for any $i = 1, \ldots, N$, we have,

$$\nabla_{x_i} J = \left(\frac{\partial x_{i+1}}{\partial x_i} \right)^\top \nabla_{x_{i+1}} J \tag{11.2.2}$$

$$\nabla_{w_i} J = \left(\frac{\partial x_{i+1}}{\partial w_i} \right)^\top \nabla_{x_{i+1}} J \tag{11.2.3}$$

We can use these relations to recursively calculate all the gradients of our system using only partial derivatives and *back propogating* gradients from later modules in the system. This process begins at the output.

Note that there are numerous variations on neural network architectures and update algorithms for domain-specific applications. Variations include pooling, probabilistic drop-out, autoregressive loss, and convolution layers, etc.

11.3 Relating LQR and Backprogation

By now, we have seen a few "backwards" algorithms in this class, the back-propagation algorithm that we just saw and the value-iteration/Riccatti recursion used for the LQR problem. One natural question one may ask is whether there are some connections between these. In fact, one can find multiple connections.

DDP, Model-based optimization and second-order backpropagation

Perhaps the most natural *policy gradient* approach is to consider optimizing the parameters of a policy where we can completely specify the dynamics and cost function as modules in a computation graph as well. In this complete, model-based case (similar to that of the LQR setting), we can use gradient descent to optimize parameters of a policy.

If we apply backpropagation to such a chain of modules (rather than a general Directed Acyclic Graph), backprogation can be understood as making a *linear* approximation of a value function. In this viewpoint, DDP can be understood as making a second order approximation of the value function. One can develop more sophisticated variants of DDP/iLQR that work on general directed graphs that can be seen as second order generalizations of backpropagation. [13].

[13]

Rederiving LQR with Back-propagation

Another connection is that we can think about the Ricatti back-up equation as coming about from following the same dynamic programming computational pattern as backpropagation, but propagating *analytic* derivatives.

Recall from the earlier lecture that the LQR problem is stated as the following,

$$\min_{u_0, \ldots, u_{T-1}} \quad \sum_{t=0}^{T-1} \left({x_t}^\top Q x_t + {u_t}^\top R u_t \right) \tag{11.3.1}$$

$$\text{s.t.} \quad x_{t+1} = A x_t + B u_t, \quad \forall t = 0, \ldots, T-2 \tag{11.3.2}$$

where $x_{t+1} = A x_t + B u_t$ is the system dynamics, and ${x_t}^\top Q x_t + {u_t}^\top R u_t$ is the instantaneous cost at each time step.

First, let us rewrite the LQR problem into a block diagram. The block diagram of the LQR problem is shown in Figure 11.3.1. Here we introduce a quadratic cost module at each time step and aggregate them into a total cost J.

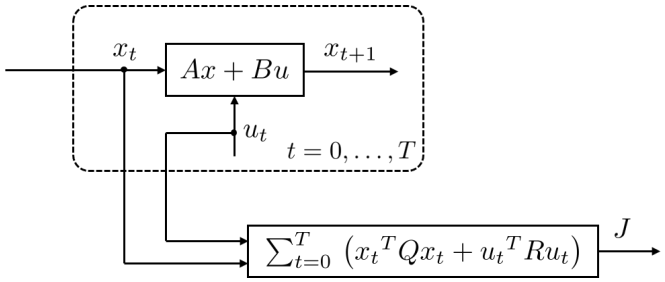


Figure 11.3.1: Finite horizon LQR realized by a block diagram.

First, we have,

$$\nabla_{u_{T-1}} J = 2R\, u_{T-1}, \tag{11.3.3}$$
$$\nabla_{x_{T-1}} J = 2Q\, x_{T-1}. \tag{11.3.4}$$

By the chain rule, for any $t = 0, \ldots, T-2$, we have,

$$\begin{aligned} \nabla_{x_t} J &= \left(\frac{\partial J}{\partial x_t}\right)^\top + \left(\frac{\partial x_{t+1}}{\partial x_t}\right)^\top \nabla_{x_{t+1}} J \\ &= 2Q\, x_t + A^\top \nabla_{x_{t+1}} J \end{aligned} \tag{11.3.5}$$

$$\begin{aligned} \nabla_{u_t} J &= \left(\frac{\partial J}{\partial u_t}\right)^\top + \left(\frac{\partial x_{t+1}}{\partial u_t}\right)^\top \nabla_{x_{t+1}} J \\ &= 2R\, u_t + B^\top \nabla_{x_{t+1}} J \end{aligned} \tag{11.3.6}$$

With the gradient we get from back-propagation, one can certainly run gradient descent for a set of controls $\{u_t\}_{t=0}^{T-1}$. The gradient descent process does not require a matrix inversion as we saw earlier, but as a cost, it requires possibly many gradient descent steps and does not provide a control policy but rather optimizes an open-loop trajectory. This can also be viewed as a policy search approach to the LQR problem.

Note, however, that we can also *solve* for optimal input using these gradients since we know that the problem is convex and we have a closed-form expression of those gradients – we can just set the gradients to zero!

- At time step $T-1$, by setting $\nabla_{u_{T-1}} J = 0$, we have,

$$2R\, u_{T-1} = 0 \qquad \Rightarrow \qquad u_{T-1} = 0. \tag{11.3.7}$$

Let $V_{T-1} = Q$, we have,

$$\nabla_{x_{T-1}} J = 2Q\, x_{T-1} \doteq 2V_{T-1}\, x_{T-1}. \tag{11.3.8}$$

- At time step $T-2$, we have,

$$\begin{aligned} \nabla_{u_{T-2}} J &= 2R\, u_{T-2} + B^\top \nabla_{x_{T-1}} J \\ &= 2R\, u_{T-2} + 2B^\top V_{T-1}\, x_{T-1} \\ &= 2R\, u_{T-2} + 2B^\top V_{T-1} (A\, x_{T-2} + B\, u_{T-2}) \\ &= 2(R + B^\top V_{T-1}\, B)\, u_{T-2} + 2B^\top V_{T-1}\, A\, x_{T-2} \end{aligned} \tag{11.3.9}$$

By setting $\nabla_{u_{T-2}} J = 0$, we have,

$$u_{T-2} = -(R + B^\top V_{T-1} B)^{-1} B^\top V_{T-1} A x_{T-2} \doteq K_{T-2} x_{T-2}. \tag{11.3.10}$$

Meanwhile,

$$\begin{aligned}
\nabla_{x_{T-2}} J &= 2Q x_{T-2} + 2A^\top \nabla_{x_{T-1}} J \\
&= 2Q x_{T-2} + 2A^\top V_{T-1} (A + B K_{T-2}) x_{T-2} \\
&= 2(Q + (A + B K_{T-2})^\top V_{T-1} (A + B K_{T-2}) \\
&\qquad - K_{T-2}^\top B^\top V_{T-1} (A + B K_{T-2})) x_{T-2} \\
&= 2(Q + (A + B K_{T-2})^\top V_{T-1} (A + B K_{T-2}) + K_{T-2}^\top R K_{T-2}) x_{T-2} \\
&\doteq 2V_{T-2} x_{T-2}.
\end{aligned} \tag{11.3.11}$$

- By repeating the process, we get,

$$\begin{aligned}
K_{t-1} &= -(R + B^\top V_t B)^{-1} B^\top V_t A \\
V_{t-1} &= Q + (A + B K_{t-1})^\top V_t (A + B K_{t-1}) + K_{t-1}^\top R_t K_{t-1}
\end{aligned} \tag{11.3.12}$$

This is precisely the Riccati equation!

11.4 Policy Gradient Methods

In the standard RL setting, we do not have access to differentiable modules that describe the dynamics (and often don't have access to the cost function in this form either). Instead, we can sample trajectories, or perhaps have access to a somewhat richer sample based model as described in earlier lectures. Value function methods like Q-learning and SARSA use information from *every* transition (s, a, r, s') in every trajectory, while black-box policy optimization methods only look at the total reward of the trajectories ignoring all structure to the reinforcement learning problem. It's natural to ask if we can use more structure without the difficulties of value function approximation.

As we have seen earlier in the lecture, if the environment model and the reward function are *known*, we can compute the policy gradient conveniently using the back-propagation algorithm. However, in reinforcement learning, we often care about the case when we don't have access to the environment model and/or the reward function. *Policy gradient methods* seek to estimate the policy gradients from trajectories without access to the environment model and the reward function.

Before we dive in to the details, we should consider whether a gradient exists for a certain policy class. This can be interpreted as a continuity condition of the mapping from the parameters in the policy class to the trajectories. This is clearly false for discrete action spaces and deterministic policies, since an infinitesimally small change in parameters can drastically change the policy and hence the trajectories. Therefore, in this lecture, we consider a class of stochastic policies parameterized by θ, $\pi_\theta : s \mapsto \pi_\theta(a|s)$. Under mild assumptions about the environment, we can safely assume that the policy gradient always exists for this policy class since stochastic policies "smooth out" the problem. [14]

Let ξ denote a *trajectory* of states and actions, $\xi = (s_0, a_0, \ldots, s_{T-1}, a_{T-1})$. We define the *total reward of the trajectory* ξ as,

$$R(\xi) = \sum_{t=0}^{T-1} r(s_t, a_t).$$

Our goal is to find the parameters that produce a policy that maximizes the expected total reward of the trajectories,

$$J(\theta) = E_{p(\xi|\theta)}[R(\xi)] = E_{p(\xi|\theta)}\left[\sum_{t=0}^{T-1} r(s_t, a_t)\right],$$

where $p(\xi|\theta)$ is the probability of the trajectory ξ given the policy parameterized by θ, which, we will see later, is also dependent on the transition model of the environment.

To find the optimal policy, we compute the policy gradient by taking the derivative with respect to θ.

$$\begin{aligned}\nabla_\theta J &= \nabla_\theta E_{p(\xi|\theta)}\left[R(\xi)\right] \\ &= \nabla_\theta \sum_{\xi \in \Xi} p(\xi|\theta)\, R(\xi),\end{aligned}$$

where Ξ denotes the set of all possible trajectories. In the case when the state and/or action space is continuous, the sum should be replaced by an

[14] The general strategy of lifting from a discrete space to a distribution to ensure continuity is used throughout machine learning and optimization. Consider, [Arora et al., 2012] as an excellent introduction to the exponentiated gradient approach to solving problems.

integral. The derivation will remain the same for integrals, although some steps would require additional justification[15].

[15] For example, the dominated convergence theorem needs to be invoked in order to swap the integral with the gradient operator in the next step.

Since $R(\xi)$ is the total reward of a *given* trajectory ξ, it has no dependence on θ. Therefore,

$$\nabla_\theta J = \sum_{\xi \in \Xi} \left(\nabla_\theta p(\xi|\theta)\right) R(\xi). \tag{11.4.1}$$

However, we cannot compute the gradient with eq. (11.4.1) because it requires us to evaluate the gradient for *all* possible trajectories. Instead, we want to obtain at least an estimate of the policy gradient using *samples* of trajectories. Therefore, we want to express the gradient as an *expectation* over probability $p(\xi|\theta)$ – the moment we do that, we can use the *law of large numbers* to draw samples from the distribution and estimate the expectation. Therefore, we use a simple trick,

$$\begin{aligned}\nabla_\theta J &= \sum_{\xi \in \Xi} \frac{p(\xi|\theta)}{p(\xi|\theta)} \left(\nabla_\theta p(\xi|\theta)\right) R(\xi) \\ &= E_{p(\xi|\theta)} \left[\frac{\nabla_\theta p(\xi|\theta)}{p(\xi|\theta)} R(\xi)\right].\end{aligned}$$

By the chain rule, we have, $\nabla_\theta \log\left(p(\xi|\theta)\right) = \frac{\nabla_\theta p(\xi|\theta)}{p(\xi|\theta)}$. So, we have an elegant expression of the policy gradient as an expectation,

$$\nabla_\theta J = E_{p(\xi|\theta)} \left[\nabla_\theta \log\left(p(\xi|\theta)\right) R(\xi)\right]. \tag{11.4.2}$$

This is sometimes called the *likelihood ratio policy gradient*. The likelihood ratio policy gradient can be interpreted as increasing the (log) probability of the trajectories with high reward and decreasing the (log) probability of the trajectories with low reward. To see this, consider a single trajectory ξ. Imagine that $R(\xi)$ is a large positive number, then if we do gradient ascent with respect to the total reward J, we are in some sense doing *gradient ascent* with respect to $\log\left(p(\xi|\theta)\right)$ according to eq. (11.4.2). Conversely, if $R(\xi)$ is a large negative number, we are performing *gradient descent* with respect to its log probability in some sense.

Note, however, that we still can not compute the policy gradient using the above equation because it requires us to evaluate $\nabla_\theta \log p(\xi|\theta)$ in the expectation, yet we do *not* know the transition model $p(s_{t+1}|a_t, s_t)$.

However, we will see that it is not a problem for policy gradient methods. If we assume the Markov property, we have,

$$p(\xi|\theta) = p(s_0) \left(\prod_{t=0}^{T-2} p(s_{t+1}|a_t, s_t)\right) \left(\prod_{t=0}^{T-1} \pi_\theta(a_t|s_t)\right).$$

Then, we have,

$$\begin{aligned}\nabla_\theta \log p(\xi|\theta) = \nabla_\theta \log p(s_0) &+ \left(\sum_{t=0}^{T-2} \nabla_\theta \log p(s_{t+1}|a_t, s_t)\right) \\ &+ \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\right).\end{aligned}$$

However, $\log p(s_0)$ and $\log p\left(s_{t+1}|s_t, a_t\right)$ do not depend on θ, so the gradients with respect to these terms are zero. Hence,

$$\nabla_\theta J = E_{p(\xi|\theta)} \left[\left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta\left(a_t|s_t\right)\right) R(\xi)\right].$$

Notice that we don't know and can't control the system dynamics, but by formulating the problem this way, we don't need to – we have control over the policy class we choose, and thus can easily compute an unbiased gradient estimate. [16] For example, we can use the *back-propagation* algorithm that we saw last week to compute the gradient $\nabla_\theta \log \pi_\theta(a_t|s_t)$.

[16] Often with outrageously high sample variance however.

As mentioned earlier, we can now use the law of large numbers to estimate this expectation,

$$\widetilde{\nabla}_\theta J = \frac{1}{N}\sum_{i=1}^{N}\left[\left(\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta\left(a_t^{(i)}|s_t^{(i)}\right)\right) R(\xi^{(i)})\right]. \tag{11.4.3}$$

By the law of large number, we know that the estimated gradient in eq. (11.4.3) is an *unbiased* estimate of the true policy gradient. Therefore, we can run *stochastic gradient ascent* with this estimated gradient. This forms the basis of the REINFORCE (Algorithm 20) algorithm (version 1, we will show some improvements soon).

Algorithm 20: The REINFORCE algorithm.

Start with an arbitrary initial policy π_θ
while *not converged* **do**
  Run simulator with π_θ to collect $\{\xi^{(i)}\}_{i=1}^N$
  Compute estimated gradient

$$\widetilde{\nabla}_\theta J = \frac{1}{N}\sum_{i=1}^{N}\left[\left(\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta\left(a_t^{(i)}|s_t^{(i)}\right)\right) R(\xi^{(i)})\right]$$

  Update parameters $\theta \leftarrow \theta + \alpha\, \widetilde{\nabla}_\theta J$
return π_θ

In step 1, we run the simulator using the current policy to collect training sequences. In step 2, we approximate the expectation by the sample mean. Step 3 is the update rule of the algorithm with α being the step size. The algorithm is then repeated until convergence or until you are bored.

An example: Tetris

We will use Tetris as an example to show how you might choose your policy function $\pi_\theta(a|s)$ and how you would compute $\nabla_\theta \log \pi_\theta(a|s)$. Suppose we have some features representing the state-action pair of the Tetris game. For instance f_1 =the number of "holes" after the placement, f_2 =the height of the highest column after the placement, etc. Due to the log in eq. (11.4.3), a convenient stochastic policy is,

$$\pi_\theta(a|s) = \frac{\exp\left(\theta^\top f(s,a)\right)}{\sum_{a'}\exp\left(\theta^\top f(s,a')\right)}.$$

This is sometimes called the Boltzmann distribution or Gibbs distribution.

The gradient of the probability distribution can be computed by any method, e.g. using back-propagation. However, it is fairly simple to solve

analytically:

$$\begin{aligned}\nabla_\theta \log \pi_\theta(a|s) &= \nabla_\theta \left[\theta^\top f(s,a) - \log \sum_{a'} \exp\left(\theta^\top f(s,a')\right)\right] \\ &= f(s,a) - \frac{\sum_{a'} f(s,a') \exp\left(\theta^\top f(s,a')\right)}{\sum_{a'} \exp\left(\theta^\top f(s,a')\right)} \\ &= f(s,a) - \sum_{a'} f(s,a')\, \pi_\theta\left(a'|s\right) \\ &= f(s,a) - E_{\pi_\theta(a'|s)}\left[f(s,a')\right]\end{aligned} \tag{11.4.4}$$

This is essentially computing the difference between the feature at state s and action a versus the expectation over all actions for that state that we could have chosen, in a way the "average" feature. Assume that we observe that feature i for action a is *larger* than the average over all actions. According to eq. (11.4.4), if performing action a at state s produces a trajectory that has high reward, we will *increase* the value of θ_i to *upweight* this particular feature. Because it seems that this feature is "helpful" for getting high rewards. On the other hand, if this state-action pair produces low reward trajectories, we may conclude that feature i is "harmful". So we make the corresponding parameter θ_i to be small or negative to reflect this observation.

11.5 *Reducing Variance*

Although the estimated gradient in eq. (11.4.3) can in theory provide an unbiased estimate, it suffers from *high variance*. In order to see this, recall that the likelihood ratio policy gradient increases the probability of the trajectories with high reward and decreases the probability of the trajectories with low reward. However, imagine when *every* trajectory has a very high reward – although some are higher than others. Then, since we only has finite number of samples at each iteration, the estimated gradient will push the probability of all these trajectories higher (if possible) since the total reward is high (and hence make the probability of other trajectories lower). However, the algorithm has no idea about the reward of trajectories *compared to other trajectories*. Therefore, we can imagine that the estimated gradients are pointing in different directions at each iteration. In fact, without making the modifications introduced in this part, the REINFORCE algorithm performs poorly compared to "black-box" approaches.

One simple modification to reduce the variance is to take advantage of causality – the actions selected now cannot affect past rewards.

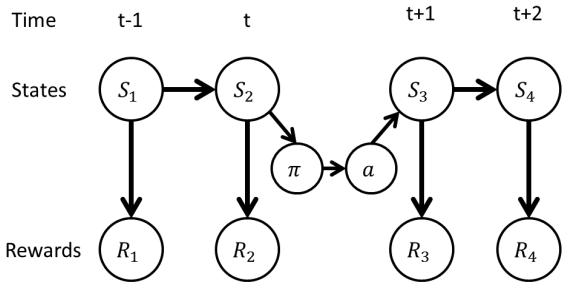


Figure 11.5.1: A trajectory of states, actions and rewards. We consider changing the action at time t in order to get a better expected future reward.

If we consider a trajectory of states and rewards, we want to change the action at time t to maximize expected reward. Intuitively, we know that changing the action at time t cannot affect the rewards obtained in the past, since we have already received them. Thus, we can represent our expected reward as only the future reward.

$$\begin{aligned}\nabla_\theta J &= E_{p(\xi|\theta)}\left[\sum_{t=0}^{T-1}\left(\nabla_\theta \log \pi_\theta(a_t|s_t)\left(\sum_{t'=0}^{t-1} r(s_{t'}, a_{t'}) + \sum_{t'=t}^{T-1} r(s_{t'}, a_{t'})\right)\right)\right] \\ &= E_{p(\xi|\theta)}\left[\sum_{t=0}^{T-1}\left(\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{t'=t}^{T-1} r(s_{t'}, a_{t'})\right)\right],\end{aligned} \tag{11.5.1}$$

where $\sum_{t'=t}^{T-1} r(s_{t'}, a_{t'})$ is sometimes called *future reward* or *reward-to-go*. We can use this idea to remove the dependence of past rewards from the calculation of our gradient.

One can reduce the variance even further by introducing *baselines* for the expected total rewards. Recall that one of the reasons for the high variance is that the algorithm does not know how well the trajectories perform *compared to other trajectories*. Therefore, by introducing a *baseline* for the total reward (or reward to go), we can update the policy based on how well the policy performs compared to a baseline. The variance can hopefully be reduced if the baseline approximates the average performance of the trajectories. But how do we know that whether the estimated gradient still makes sense?

Let's first take a look at the expectation $E_{p(\xi|\theta)}[\nabla_\theta \log p(\xi|\theta) b]$. We have,

$$\begin{aligned}E_{p(\xi|\theta)}[\nabla_\theta \log p(\xi|\theta)\, b] &= \sum_{\xi \in \Xi} \nabla_\theta\, p(\xi|\theta)\, b \\ &= \nabla_\theta \left(\sum_{\xi \in \Xi} p(\xi|\theta)\right) b \\ &= (\nabla_\theta 1)\; b = 0.\end{aligned} \tag{11.5.2}$$

Therefore, the estimated policy is still unbiased if we introduce a baseline for the total reward (or reward to go). Note here that the above equation holds as long as b does not depend on θ, hence b can potentially be a function of the state, i.e. $b = b(s_t)$.[17] In fact, a common choice of baseline is the value function or some estimate of the value function.

[17] However, some additional effort is needed to show that a time-dependent baseline actually works, including expanding $p(\xi|\theta)$ in the expectation as a product of the transition probability and the policy.

Putting everything together, we can generate another policy gradient expression,

$$\nabla_\theta J = E_{p(\xi|\theta)}\left[\sum_{t=0}^{T-1}\left(\nabla_\theta \log \pi_\theta(a_t|s_t)\left(\sum_{t'=t}^{T-1} r(s_{t'}, a_{t'}) - b(s_t)\right)\right)\right], \tag{11.5.3}$$

We estimate the above policy gradient as

$$\widetilde{\nabla}_\theta J = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T-1}\left(\nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)})\left(\sum_{t'=t}^{T-1} r(s_{t'}^{(i)}, a_{t'}^{(i)}) - b(s_t^{(i)})\right)\right). \tag{11.5.4}$$

This can give us an unbiased estimate of the policy gradients with lower variance.

11.6 Eligibility Traces

Conveniently, the approach described above can be effectively implemented with a simple infinite impulse response filter, rather than by remembering entire trajectories. To lighten notation, consider the case when no baselines are introduced, i.e. $b \equiv 0$.

Given a trajectory, we can introduce an iteratively computed *eligibility vector*,

$$e_t = e_{t-1} + \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Note then that,

$$e_t \cdot r(s_t, a_t) = \sum_{t'=0}^{t} \nabla_\theta \log \pi_\theta(a_{t'}|s_{t'})\, r(s_t, a_t).$$

We will see that the gradient then is just a running sum of the expected future rewards over all visited states at each time-step,

$$\Delta_t = \Delta_{t-1} + e_t \cdot r(s_t, a_t).$$

If we expand this out, we can see that it is the same as gradient calculated by the likelihood ratio method.

$$\begin{aligned}\Delta_t = \nabla_\theta \log \pi_\theta(a_0|s_0)\, r(s_0, a_0) + \sum_{t=0}^{1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, r(s_1, a_1) + \\ \cdots + \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, r(s_T, a_T) \\ = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T-1} r(s_{t'}, a_{t'})\end{aligned}$$

11.7 REINFORCE

The REINFORCE algorithm uses the eligibility trace to calculate the gradient update. We start off with a set or parameters and several trajectories gathered by forward simulating the policy generated by those parameters. We can then use the eligibility trace to calculate the gradient and get a new set of parameters. We add a discount factor, γ, to manage the general class of infinite horizon discounted problems.

Algorithm 21: REINFORCE Algorithm

1: $e = 0$
2: $\Delta = 0$
3: **for all** t **do**
4: $\quad e \leftarrow \gamma\, e + \nabla_\theta \log \pi_\theta(a_t|s_t)$
5: $\quad \Delta \leftarrow \Delta + \frac{1}{t+1}\left[r(s_t, a_t) \cdot e - \Delta\right]$
6: **end for**

The resulting Δ_{t+1} is a noisy estimate of the gradient. We can either compute Δ_{t+1} several times to get a less noisy estimate, or we can move a small amount using the noisy estimate.

11.8 The Policy Gradient Theorem

The REINFORCE algorithm calculates the gradient using expected future reward as determined by a trajectory.

$$\nabla_\theta J = E_{p(\xi|\theta)} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T-1} r(s_{t'}, a_{t'}) \right]$$

We can instead replace the the estimate of future reward $\sum_{t'=t}^{T-1} r(s_{t'}, a_{t'})$ with the action value Q^{π_θ}, which by definition gives us the expected future reward.

$$\nabla_\theta J = E_{p(\xi|\theta)} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, Q^{\pi_\theta}(s_t, a_t) \right]$$

We can update the gradient rule to take the expectation over the distribution of *states* rather than the expectation over the *trajectories*, this leads to the *Policy Gradient Theorem*.

$$\nabla_\theta J = E_{s\sim d^{\pi_\theta}(s), a\sim\pi_\theta(a|s)} \left[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s,a) \right] \tag{11.8.1}$$

Here, $d^{\pi_\theta}(s)$ is the distribution of states under policy π_θ, i.e., the fraction of time spent in state s,

$$d^{\pi_\theta}(s) = \frac{1}{T} \sum_{t=0}^{T-1} p^{\pi_\theta}(s, t),$$

where $p^{\pi_\theta}(s,t)$ is the probability that state s is visited at step t under policy π_θ.

The policy gradient theorem states that the gradient of average reward under a policy π_θ parametrized by θ is given by

$$\nabla_\theta J = E_{d^{\pi_\theta}(s)} E_{\pi_\theta(a|s)} \left[\nabla_\theta \log(\pi_\theta(a|s)\, Q^{\pi_\theta}(s,a) \right] \tag{11.8.2}$$

The expectations are with respect to the distribution $d^{\pi_\theta}(s)$ of states given a policy π_θ and the actions taken under the policy π_θ given the state s. We can prove that, for the value function $V^{\pi_\theta}(s)$ is only a function of the state s, it can viewed as a *baseline* as we saw above. Thus, Eq. 11.8.2 is equal to:

$$\nabla_\theta J = E_{d^{\pi_\theta}(s)} E_{\pi_\theta(a|s)} \left[\nabla_\theta \log(\pi_\theta(a|s)\, (Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)) \right], \tag{11.8.3}$$

where $A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$ is referred to as the *advantage* of action a at state s under policy π_θ. So why is this true?First, consider the inner expectation. Because V^{π_θ} does not depend on a, this is equivalent to,

$$E_{\pi_\theta(a|s)} \left[\nabla_\theta \log(\pi_\theta(a|s) V^{\pi_\theta}(s) \right] = V^{\pi_\theta}(s)\, E_{\pi_\theta(a|s)} \left[\nabla_\theta \log(\pi_\theta(a|s)) \right]. \tag{11.8.4}$$

That leaves $\nabla_\theta \log(\pi_\theta(a|s)$ in the expectation. Intuitively that must be equal to zero because the probability distribution π_θ must sum to one, so the sum over all changes must be equal to zero. We show more explicitly below that this is indeed the case. We expand (Eq. 11.8.4) into sums over the states

and actions. We can show that,

$$\begin{aligned} E_{\pi_\theta(a|s)}\left[\nabla_\theta \log(\pi_\theta(a|s)\right] &= \sum_{a\in\mathbb{A}} \pi_\theta(a|s)\, \nabla_\theta \log(\pi_\theta(a|s)) \\ &= \sum_{a\in\mathbb{A}} \pi_\theta(a|s)\, \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \\ &= \sum_{a\in\mathbb{A}} \nabla_\theta \pi_\theta(a|s) \\ &= \nabla_\theta \left(\sum_{a\in\mathbb{A}} \pi_\theta(a|s) \right) = \nabla_\theta 1 = 0. \end{aligned} \tag{11.8.5}$$

Through linearity of expectation, we have,

$$\begin{aligned} & E_{d^{\pi_\theta}(s)} E_{\pi_\theta(a|s)}\left[\nabla_\theta \log(\pi_\theta(a|s) V^{\pi_\theta}(s)\right] \\ =& E_{d^{\pi_\theta}(s)}\left[V^{\pi_\theta}(s)\, E_{\pi_\theta(a|s)}\left[\nabla_\theta \log(\pi_\theta(a|s)\right]\right] \\ =& E_{d^{\pi_\theta}(s)}\left[V^{\pi_\theta}(s)\cdot 0\right] = 0. \end{aligned} \tag{11.8.6}$$

Finally,

$$\begin{aligned} \nabla_\theta J &= E_{d^{\pi_\theta}(s)} E_{\pi_\theta(a|s)}\left[\nabla_\theta \log(\pi_\theta(a|s) Q^{\pi_\theta}(s,a)\right] \\ &= E_{d^{\pi_\theta}(s)} E_{\pi_\theta(a|s)}\left[\nabla_\theta \log(\pi_\theta(a|s)\, (Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s))\right] \\ &= E_{d^{\pi_\theta}(s)} E_{\pi_\theta(a|s)}\left[\nabla_\theta \log(\pi_\theta(a|s) A^{\pi_\theta}(s,a)\right] \end{aligned} \tag{11.8.7}$$

Intuitively, this shows that the algorithm wants the advantage of the action to be high, and wants to choose actions that are correlated with the advantage being high. It adjusts the policy by making small changes towards Q values that are higher than the average.

The policy gradient theorem connects estimating the gradient $\nabla_\theta J$ with estimating Q^{π_θ} or A^{π_θ}. For example,we can estimate Q^{π_θ} with some parameterized function $Q_\phi^{\pi_\theta}$ using *Approximate Dynamic Programming* methods like Fitted Q-Iteration or an advantage estimator $A_\phi^{\pi_\theta}$, to approximate the advantage function $A^{\pi_\theta}(s,a)$. Also, we can use samples trajectories under policy π_θ to estimate the expectation in Eq. (11.8.7), which results in an estimated policy gradient,

$$\widetilde{\nabla}_\theta J = \frac{1}{N} \sum_{i=1}^{N} \left(\nabla_\theta \log \pi_\theta(a_i|s_i)\, A_\phi^{\pi_\theta}(s_i,a_i) \right). \tag{11.8.8}$$

This leads to a class of methods called *Actor–Critic Methods*. Actor–Critic methods learn a *actor* (the policy) and a *critic* simultaneously. The critic produces the estimate of some value function (e.g., state-value function, action-value function, advantage function, etc.) for bootstrapping (updating the value function estimate for a state from the estimated values of other states). By introducing the critic, the variance of the gradient estimate can be further reduced. Many popular policy gradient algorithms, including TRPO, PPO and DDPG, adopt the actor–critic architecture.

Examples

Let us consider a simple example of the actor-critic algorithm. Say we have two actions that we can take from a given state and one feature for the state s. One of our actions a_0 is bad, while the other one a_1 is good.

We use the Boltzmann distribution that we have seen in the previous example,

$$\pi_\theta(a|s) = \frac{\exp[\theta^\top f(s,a)]}{\sum_{a'} \exp[\theta^\top f(s,a)]}.$$

Suppose that the features of our state and the two actions are $f(s,a_0) = 3$ and $f(s,a_1) = 1$.

Let's say our current value of the parameter θ is $\theta = 1$. Then, the probabilities for taking each action are,

$$\pi_\theta(a_0|s) = \frac{\exp[\theta^\top f(s,a_0)]}{\exp[\theta^\top f(s,a_0)] + \exp[\theta^\top f(s,a_1)]} = \frac{e^3}{e^3+e} = \frac{e^2}{e^2+1} \approx 0.88,$$

$$\pi_\theta(a_1|s) = \frac{\exp[\theta^\top f(s,a_1)]}{\exp[\theta^\top f(s,a_0)] + \exp[\theta^\top f(s,a_1)]} = \frac{e}{e^3+e} = \frac{1}{e^2+1} \approx 0.12,$$

where $e \approx 2.71828$ is the base of the natural logarithm

We then get an estimate of the future reward, possibly through our critic: $Q^\pi(s,a_0) = 1$ and $Q^\pi(s,a_1) = 100$.

We have already seen previously that we can compute the derivative of the log probability as follows:

$$\nabla_\theta \log \pi_\theta(a|s) = f(s,a) - E_{\pi_\theta(a'|s)}[f(s,a')],$$

where,

$$E_{\pi(a'|s)}[f(s,a')] \approx 0.88 \times 3 + 0.12 \times 1 = 2.76.$$

We can just compute the gradient estimate[18],

$$\begin{aligned}
\widetilde{\nabla}_\theta J &= E_{a\sim\pi_\theta(a|s)}\left[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s,a)\right] \\
&= \pi_\theta(a_0|s)\nabla_\theta \log \pi_\theta(a_0|s) Q^{\pi_\theta}(s,a_0) \\
&\qquad + \pi_\theta(a_1|s)\nabla_\theta \log \pi_\theta(a_1|s) Q^{\pi_\theta}(s,a_1) \\
&\approx 0.88 \times (3 - 2.76) \times 1 + 0.12 \times (1 - 2.76) \times 100 \\
&\approx -20.79
\end{aligned}$$

[18] Note that this is an estimate because we are not taking expectation over state

Thus the policy gradient algorithm tells us to decrease the value of θ since the higher feature value seems to result in lower future reward. This makes the probability of choosing a_1 at s higher than the previous iteration.

11.9 Highly Correlated Features

Gradient ascent/descent methods depend greatly on the *parameterization* of the policy. To see this, consider the two parameterizations of Tetris.

Parameterization 1: $f_1 =$ # of Holes after the placement, $f_2 =$ Height after the placement. We use θ to denote the parameter for this parameterization.

Parameterization 2: $g_1 = \ldots = g_{100} =$ # of Holes after the placement, $g_{101} =$ Height after the placement. We use ϕ to denote the parameter for this parameterization

Then, for Parameterization 1, we have,

$$\theta^\top f(x,a) = \theta_1 \times \#\text{ of Holes}(x,a) + \theta_2 \times \text{Height}(x,a).$$

While for Parameterization 2, we have,

$$\phi^\top g = \left(\sum_{i=1}^{100} \phi_i\right) \times \#\text{ of Holes}(x,a) + \phi_{101} \times \text{Height}(x,a).$$

When we take the policy gradient, we have,

$$\nabla_{\theta_i} J = E_{p(\xi|\theta)}\left[\sum_{t=0}^{T-1}\left(f_i(s,a) - E_{\pi_\theta(a'|s)}[f_i(s,a')]\right) Q^{\pi_\theta}(s_t,a_t)\right]$$
$$\nabla_{\phi_i} J = E_{p(\xi|\phi)}\left[\sum_{t=0}^{T-1}\left(g_i(s,a) - E_{\pi_\phi(a'|s)}[g_i(s,a')]\right) Q^{\pi_\phi}(s_t,a_t)\right]$$

Hence, we have $\nabla_{\phi_1} J = \ldots = \nabla_{\phi_{100}} J = \nabla_{\theta_1} J$. The policy gradient algorithm takes a 100 times larger step for the *actual weight* corresponding to the number of holes using Parametrization 2 than in Parametrization 1!

Gradient ascent (or steepest ascent) poses the problem of finding $\max_{\Delta\theta} J(\theta + \Delta\theta)$ such that $\delta\theta$ is small. Gradient ascent measures "small" as the l_2 norm $\|\Delta\theta\|_2 = \sqrt{\sum_i (\Delta\theta_i)^2} \leq \epsilon$. However this version of measuring "small" depends on the parameterization of our policy. Ideally, we want the descent to measure "small" based on changes in our policy and not depend on the parameterization of the policy. We will address this problem in the next lecture.

11.10 Natural Policy Gradient

In the general formulation of steepest descent, as given by Eq. (11.10.1), there are many size metrics that can be utilized.

$$\max_{\Delta\theta} J(\theta + \Delta\theta) \qquad s.t. \qquad \|\Delta\theta\| \leq \epsilon \tag{11.10.1}$$

The gradient descent algorithm comes about when we choose the metric $\|\cdot\|$ to be the l_2 norm over the parameters ($\sqrt{\Delta\theta^\top \Delta\theta}$). In policy gradient methods such as REINFORCE, this definition of the metric can cause the algorithm to fail, if utilizing highly correlated features. This is due to the fact that the l_2 norm defines a "small" change in the gradient direction as depending on the cumulative sum in parameter change, which may have varying degrees of correlation with actual policy change. Instead, we would like to define the size metric such that the notion of "small" encompasses changes in the parameterized *policy*, not simply the changes in the parameters themselves. This leads to two questions.

Q1) What does steepest descent look like given other metrics?

Q2) What metric captures the fact that we would like our metric to be tied to the difference between the $\pi_\theta(a|s)$ and $\pi_{\theta+\Delta\theta}(a|s)$, and not just θ and $\theta + \Delta\theta$?

Q1 – What does steepest descent look like under other metrics?

For small changes in the parameters, we can think of the metric as some quadratic function of the parameters, as evidenced by the Taylor expansion. The steepest descent optimization problem then becomes

$$\max_{\Delta\theta} J(\theta + \Delta\theta) \qquad s.t. \qquad \Delta\theta^\top G(\theta)\Delta\theta \leq \epsilon \tag{11.10.2}$$

where $G(\theta)$ defines the specific metric. In general, G is a distance metric and thus is symmetric positive semi-definite[19]. This matrix defines the notion of distance in the parameter space locally around θ and, in some cases, can be constant; if this is true, the metric is referred to as flat. Intuitively, a flat metric entails that distance is measured the same everywhere in the parameter space. While a flat metric can be helpful, in the general case it will not accurately capture the true notion of distance on the parameter manifold.

[19] Being pedantic, it is actually a pseudo-metric if it has nontrivial nullspace

However, we don't always want to use flat metrics because it does not always precisely reflect what does "small" means in our particular situations. For example, one change of parameters $\Delta\theta$ at θ_1 can result in a very minor change of our policy, while the same $\Delta\theta$ can result in a large change at θ_2. We want our metric $G(\theta)$ to reflect that.

We can solve this new optimization problem (Eq. (11.10.2)) for the parameters using the technique of Lagrange multipliers. This converts the constrained optimization problem (11.10.2) to unconstrained optimization problem with respect to the *Lagrangian* of the system,

$$\max_{\Delta\theta} \mathcal{L}(\Delta\theta, \lambda) = J(\theta + \Delta\theta) - \lambda \left[\Delta\theta^\top G(\theta)\Delta\theta - \epsilon\right], \tag{11.10.3}$$

where $\lambda \geq 0$ is the *Lagrange multiplier*.

The theory says that there exists a choice of $\lambda \geq 0$ such that the constraint optimization problem (11.10.2) and the unconstrained optimization problem

(11.10.3) has the same solution. To begin with, we could use only the direction and simple take λ to be a fixed scalar and solve for $\Delta\theta$. However, it's been demonstrated in practice that parameterizing in terms of ϵ is actually a good way to control step-size. Note that this is straightforward to compute since we can simply normalize $\delta theta$ to ϵ norm in the G metric (that is, it's easy to explicitly compute the correct lagrange multiplier)! [20]

[20] This is the dominant benefit of the "TRPO" method over naive implementations of the natural gradient.

Because we are only considering small steps in $\Delta\theta$, we can approximate (11.10.3) by using the first-order Taylor expansion of J:

$$\mathcal{L}(\Delta\theta, \lambda) \approx J(\theta) + \Delta\theta^\top \nabla_\theta J - \lambda \left[\Delta\theta^\top G(\theta)\Delta\theta - \epsilon\right] \doteq \tilde{\mathcal{L}}_\lambda(\Delta\theta). \tag{11.10.4}$$

Here we use the notation $\tilde{\mathcal{L}}_\lambda(\Delta\theta)$ to emphasize that we are taking λ as a constant and hence $\tilde{\mathcal{L}}_\lambda$ is a function of $\Delta\theta$.

Note that the approximated Lagrangian $\tilde{\mathcal{L}}_\lambda$ is quadratic in $\Delta\theta$. To find the solution, we can simply take the partial derivative of the approximated Lagrangian $\tilde{\mathcal{L}}_\lambda$ with respect to the change in parameters and set it to zero:

$$\frac{\partial \tilde{\mathcal{L}}_\lambda}{\partial \Delta\theta} = \nabla_\theta J - 2\,\lambda\, G(\theta)\,\Delta\theta = 0. \tag{11.10.5}$$

If $G(\theta)$ is nonsingular, the solution to the above equation is thus:

$$\Delta\theta = \frac{1}{2\,\lambda}\, G^{-1}(\theta)\, \nabla_\theta J. \tag{11.10.6}$$

Intuitively, we are taking the gradient and multiplying it by the inverse of the metric that defines what it means to be large, and then taking a step in that direction. However, it may still be the case that $G(\theta)$ is singular, or very close to singular, due to two features being very highly correlated. For example, if we are using two features that are exactly the same, the metric should look something like

$$G(\theta) = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}.$$

In this case, if we make change $\Delta\theta = [\Delta\theta_1\ \Delta\theta_2]^\top$ to the parameters, the size of this change measured by metric $G(\theta)$ is thus,

$$\begin{aligned} \Delta\theta^\top G(\theta)\Delta\theta &= \begin{bmatrix} \Delta\theta_1 & \Delta\theta_2 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \end{bmatrix} \\ &= \Delta\theta_1^2 + 2\Delta\theta_1\Delta\theta_2 + \Delta\theta_2^2 \\ &= (\Delta\theta_1 + \Delta\theta_2)^2 \end{aligned}$$

This means that changes in any of the features, or any combination of the features should be the same if they add up to be the same because they effectively act on the same feature. Because this matrix is singular, there exists a space in which we can move, and it will not change the policy at all (the nullspace of $G(\theta)$). For example, we can add δ to the first parameter and subtract the second by δ, and the policy is still the same. In this case, the most natural thing to do is use the pseudo-inverse, denoted as $G^\dagger(\theta)$, in place of the inverse, which means that we not trying to do anything in the nullspace, only the space in which we can actually affect things.

Q2 – What metric do we want to use for policy gradients?

Despite now knowing how to change and solve the optimization problem for different metrics, we are still left with the question of what the metric should be. It turns out that there is a canonical answer for probability distributions, given by Chentsov's theorem. This theorem effectively says that there is a unique metric such that distance is invariant to a class of changes to the problem, such as label switching, for parametric family of distributions; this metric is known as the *Fisher Information Metric* (Eq. 11.10.7).

$$G(\theta) = E_{p_\theta}\left[\nabla_\theta \log(p_\theta)\, \nabla_\theta \log(p_\theta)^\top\right] \tag{11.10.7}$$

Another way to come to this same result is to consider the Kullback–Leibler divergence, or K-L divergence, of two probability distributions. Given two probability distributions p and q,

$$\mathcal{KL}(p\|q) = \sum_{x\in\mathbb{X}} p(x) \log\left(\frac{p(x)}{q(x)}\right). \tag{11.10.8}$$

It turns out that the change in parameters measured by the Fisher Information Metric is exactly the second order approximation of the K-L divergence of the probability distributions before and after the change,

$$\begin{aligned} \mathcal{KL}(p_{\theta+\Delta\theta}\|p_\theta) &\approx \Delta\theta^\top G(\theta)\Delta\theta, \\ \mathcal{KL}(p_\theta\|p_{\theta+\Delta\theta}) &\approx \Delta\theta^\top G(\theta)\Delta\theta. \end{aligned} \tag{11.10.9}$$

In general, the second-order approximation of "obvious" metrics on probability distributions will result in the Fisher Information Metric.

For the specific problem of policy optimization, we take the Fisher Information Metric on trajectories as our metric (Eq. 11.10.10). This is because we want to essentially measure the distance between trajectories (distributions of states) given changes in parameters.

$$G(\theta) = E_{d^{\pi_\theta}(s),\pi_\theta(a|s)}\left[\nabla_\theta \log \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s)^\top\right], \tag{11.10.10}$$

Recall that $d^{\pi_\theta}(s)$ is the distribution of states, or the fraction of time spent in states, under policy π_θ.

In practice, $G(\theta)$ can be estimated as a running average of the states experienced (Eq, 11.10.11), and its inclusion makes an enormous difference in the success of algorithms such as REINFORCE. [21]

$$\tilde{G}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\nabla_\theta \log \pi_\theta(a_i|s_i) \nabla_\theta \log \pi_\theta(a_i|s_i)^\top\right] \tag{11.10.11}$$

Intuitively, from a Machine Learning perspective, this algorithm is attempting to move in the direction that improves the performance the most, subject to changing the distribution of input examples as little as possible. This is also very similar to whitening of data, a natural normalization technique in Machine Learning.

In the general case, where we are just doing steepest descent with a distance metric, the algorithm is referred to as the covariant gradient method. In the special case shown above when you are measuring is distance between probability distributions, the algorithm is known as the natural gradient method.

[21] J. A. Bagnell and J. Schneider. Covariant policy search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2003

Then, we can combine this estimated policy gradient with the natural gradient method, which gives us the update rule,

$$\Delta\theta = \frac{1}{2\lambda}\,\tilde{G}^{-1}(\theta)\,\widetilde{\nabla}_\theta J. \tag{11.10.12}$$

This is known as the *Natural Policy Gradient* method. Note that Eq. (11.10.12) requires inverting the estimated Fisher information matrix, which can be computationally expensive when the number of parameters is large. One solution is to solve for Eq. (11.10.12) through iterative methods, e.g., Conjugate Gradient method, and terminate early. This in practice gives us reasonably good estimates of the natural policy gradient.

11.11 Conservative Policy Iteration

REINFORCE is essentially like a soft policy iteration, trying to change the probability of actions so that they are correlated with things that have high Q values. However, REINFORCE does not suffer from the disadvantages of policy iteration, because it makes small changes.

We can modify approximate policy iteration to avoid the problems caused by making big changes at each time step. We can make the policy iteration stochastic, by choosing to follow the old policy with probability α, and taking action $\text{argmax}_a \tilde{Q}(s,a)$ with probability $1-\alpha$. This algorithm, known as *conservative policy iteration*, essentially makes a small change to the probability distribution over trajectories, but by choosing actions to go the steepest direction uphill.

11.12 Related Reading

[1] McNamara, A., Treuille, A., Popović, Z. and Stam, J., *Fluid control using the adjoint method*, ACM Transactions On Graphics (TOG) 2004.

[2] Krizhevsky, A., Sutskever, I. and Hinton, G.E., *ImageNet Classification with Deep Convolutional Neural Networks*, NIPS 2012.

[3] Le, Quoc V, *Building high-level features using large scale unsupervised learning*, Acoustics, Speech and Signal Processing (ICASSP), 2013.

[4] Bagnell, J.A. and Schneider, J. *Covariant policy search*, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 2003.

12
Iterative Learning Control

(This lecture is related to the paper **Using Inaccurate Models in Reinforcement Learning** [1]. Reading the paper first is helpful in understanding the material).

In the previous lectures, we have been looking into using policy gradient methods to find a good policy. The main advantage of policy gradient methods is that they require us to know *very little* about the problem we are solving – we don't need to know the transition model, and we don't need to know the reward function either. All we need to do is collect a number of roll-out trajectories and estimate the policy gradient based on that. As a result of that, however, policy gradient methods have their natural limitation that they generally require *a large number* of trajectories to work reasonably well and they sometimes suffer from high variance in their gradient estimates.

In this lecture, we take a different path by assuming that we know *something* about the particular problem we are solving. In particular, we assume that we have a possibly inaccurate but hopefully helpful model of the system and we know the reward function. We will see how we can approach the reinforcement learning problem differently with this additional knowledge.

12.1 Model-based Reinforcement Learning

One straightforward way to solve this problem is to find an optimal policy with respect to the (possibly inaccurate) model that we already have. This idea lays the foundation of model-based reinforcement learning and optimal control. In fact, we have seen an example of model-based reinforcement learning techniques earlier in this class – LQR. It solves for the optimal policy for a linear model and a quadratic reward function – although any practical systems are hardly truly linear.

Given a (possibly time-varying) deterministic model $\hat{f}_t : \mathbb{S} \times \mathbb{A} \to \mathbb{S}$ and a reward function $r : \mathbb{S} \times \mathbb{A} \to \mathbb{R}$, there is a slightly more general way to find a good policy: solve for the policy gradient through back-propagation[1].

[1] Or the adjoint method if you are an optimal controls person.

Assume that we parameterize our policy π_θ with parameter θ. Then, we can represent the model-based reinforcement learning as a block diagram:

Note that we are using $\hat{s}_t$, $\hat{a}_t$ and $\hat{J}$ here because they are *not* the actual state, action and total reward we would get from running the actual system, rather, they are just what we get from simulating through our approximated model $\hat{f}_t$. Note, however, that s_0 is not approximated because we assume that

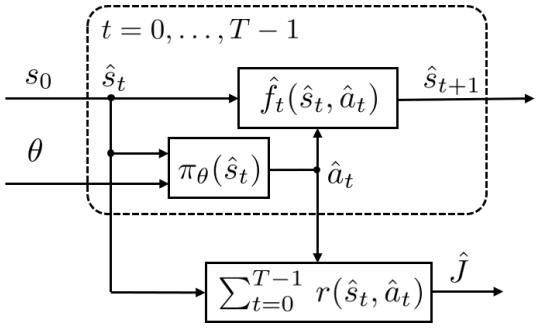


Figure 12.1.1: The block diagram representation of a model-based RL problem using an approximated model.

we start from a fixed state.

Recall from the earlier lecture that we can optimize the policy through the "forward-propagation, back-propagation, gradient ascent" scheme. We initialize our parameter at some arbitrary $\theta^{(0)}$. At i-th iteration, we do:

1. **Forward-propagation:** Forward simulate $\pi_{\theta^{(i)}}$ using the *approximated model* $\hat{f}_t$ and observe the simulated trajectory $\{s_0, a_0^{(i)}\}, \{\hat{s}_1^{(i)}, \hat{a}_1^{(i)}\}, \cdots \{\hat{s}_T^{(i)}, \hat{a}_T^{(i)}\}$ along with the approximated total reward $\hat{J}$.
2. **Back-propagation:** Compute the approximated policy gradient $\nabla_\theta \hat{J}(\theta)$ along the trajectory $\{s_0, a_0^{(i)}\}, \{\hat{s}_1^{(i)}, \hat{a}_1^{(i)}\}, \cdots \{\hat{s}_T^{(i)}, \hat{a}_T^{(i)}\}$ using back-propagation.
3. **Gradient-Ascent:** Update the parameter $\theta^{(i+1)} = \theta^{(i)} + \alpha \nabla_\theta \hat{J}(\theta)$.

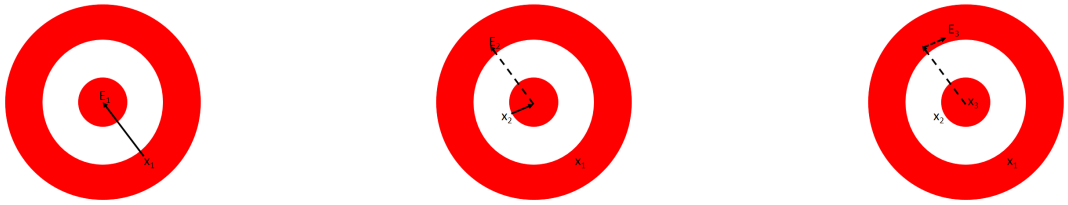
Note that the approximated policy gradient $\nabla_\theta \hat{J}(\theta)$ we used here is fundamentally different from the estimated policy gradient $\widehat{\nabla}_\theta J(\theta)$ we used for the policy gradient methods. Here we use $\nabla_\theta \hat{J}(\theta)$, which is the *exact* gradient of the *approximated* total reward function, while $\widehat{\nabla}_\theta J(\theta)$ is the *estimated* gradient for the *exact* total reward function.

12.2 *Iterative Learning Control*

A typical problem in model-based reinforcement learning is that no matter how well you try to model the system dynamics, there are always unmodeled errors that can easily throw your controller off course. A concrete example is given in [1], where the authors want to control an RC car to follow some trajectory. It is shown that the carpet threading is enough to cause their linearized system to drift away from the planned trajectory.

In contrast, it is against our human intuition that a super sophisticated model is required to perform many tasks such as steering a car. A young adult who only has a crude idea of how a car steers can learn to make good turns after a few trials: if the turn is too wide, steer more next time; if the turn is too tight, steer less next time. This idea is also illustrate in the target example in Fig 12.2.1.

Thus the key idea of Iterative Learning Control is (as the authors state): "... to use a real world trial to *evaluate* a policy but then use the simulator (or model) to estimate the *derivative* of the evaluation with respect to the policy parameters." *In other words, we can use the actual system to do forward propagation and then use our crude model for back propagation.*



(a) Initial shot is off (b) Aim at E_2 instead of E_1 (c) Aim at E_3 and hit the bull's eye

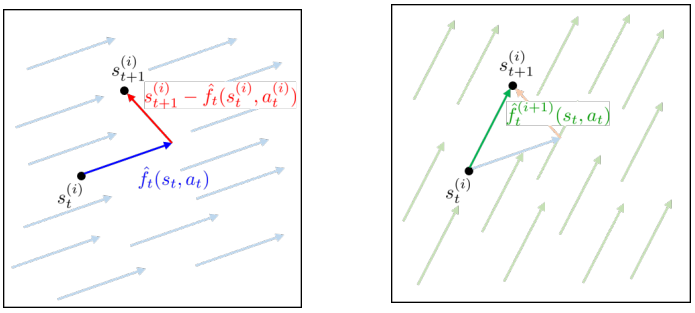
Figure 12.2.1: The target example: (a) Initially we aim at the bull's eye(E_1) but due to wind or miscalibrated sight we end up at x_1. (b) Instead of aiming at E_1 which will end up at x_1, we aim at E_2 which hopefully will end up at E_1. (c) Continue updating the offset until we hit the bull's eye.

The Algorithm

Consider an approximated MDP problem (approximated in a sense that the model isn't very accurate but still informative) $(\mathbb{S}, \mathbb{A}, \hat{f}_t, s_0, r)$, where $\mathbb{S}$ is the set of all possible states, A is the set of all actions, $\hat{f}_t : \mathbb{S} \times \mathbb{A} \to \mathbb{S}$ is the (possibly time-varying) deterministic approximated transition model, s_0 is the initial state and $r : \mathbb{S} \to \mathbb{R}$ is the reward function[2]. Assume both the system and the policy are deterministic and the policy is parameterized by θ. We initialize our parameter at $\theta^{(0)}$, the solution to the model-based reinforcement learning problem we saw in the previous part. Then the i-th iteration of the policy gradient proceeds as follows:

1. Execute the current policy $\pi_{\theta^{(i)}}$ on the *real system* and observe the actual trajectory $\{s_0, a_0^{(i)}\}, \{s_1^{(i)}, a_1^{(i)}\}, \cdots \{s_T^{(i)}, a_T^{(i)}\}$.
2. Augment the model by adding a (time-dependent) bias term to the original model at every time step t: $\hat{f}_t^{(i+1)}(s,a) = \hat{f}_t(s,a) + \left(s_{t+1}^{(i)} - \hat{f}_t(s_t^{(i)}, a_t^{(i)})\right)$.
3. Compute policy gradient $\nabla_\theta J(\theta)$ using back-propagation with the updated model and then update the parameter $\theta^{(i+1)} = \theta^{(i)} + \alpha \nabla_\theta J(\theta)$.

In each iteration i, adding the time-dependent bias terms corrects the old model so that if we re-run it with $\pi_{\theta^{(i)}}$ and $\hat{f}^{(i+1)}(s,a)$ we would get the exact same state-action sequence $\{s_0, a_0^{(i)}\}, \{s_1^{(i)}, a_1^{(i)}\}, \cdots \{s_T^{(i)}, a_T^{(i)}\}$.



(a) The original model $\hat{f}_t$. (b) The augmented model $\hat{f}_t^{(t+1)}$.

Figure 12.2.2: At each iteration, we augment the model by adding a (time-dependent) bias term to the original model so that we would get the same trajectory.

Therefore when updating the parameters θ in step 3, the correct trajectory is used for computing the policy gradient. In most nonlinear control systems,

[2] Note that here that we assume that we know the *true* reward function. Note also that the reward is only defined on states in this paper but one can also define it as a function of both states and actions.

this means using the actual trajectory for the linearization points though the derivatives are computed using the old model (bias terms do not affect derivatives) at these correct trajectory points.

12.3 The Theory

Once again we assume the system is deterministic and assume our policy is parameterized by θ. Define the following function $s_t = h_t(s_0, \theta)$:

$$h_1(s_0, \theta) = s_1 = f_0(s_0, \pi_\theta(s_0)) \tag{12.3.1}$$
$$h_t(s_0, \theta) = f_{t-1}(s_{t-1}, \pi_\theta(s_{t-1})) \tag{12.3.2}$$
$$= f_{t-1}(h_{t-1}(s_0, \theta), \pi_\theta(h_{t-1}(s_0, \theta))) \tag{12.3.3}$$

In other words, $h_t(s_0, \theta)$ is the *real world* state at time t if we start at s_0 and follow the policy π_θ. Similarly we can define $\hat{s}_t = \hat{h}_t(s_0, \theta)$ which is the state at time t using the approximated model and following π_θ.

Let $s_0, s_1, \cdots s_T$ be the *real world* state sequence obtained when executing the policy π_θ. Then the true policy gradient is given by:

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T} \nabla_{s_t} r(s_t) \frac{dh_t}{d\theta}\Big|_{s_0, s_1, \cdots s_{T-1}} \tag{12.3.4}$$

Note here that the derivatives $\frac{dh_t}{d\theta}$ are total derivatives since h_t is dependent on θ through all previous time steps $t' = 0, \ldots, t-1$. The chain rule (back-propagation) is applied to every term in $\frac{dh_t}{d\theta}$ by the definition of Eq. 12.3.3.

Similarly we can define the approximated policy gradient as follows:

$$\nabla_\theta \hat{J}(\theta) = \sum_{t=0}^{T} \nabla_{\hat{s}_t} r(\hat{s}_t) \frac{d\hat{h}_t}{d\theta}\Big|_{s_0, \hat{s}_1, \cdots \hat{s}_{T-1}} \tag{12.3.5}$$

Two sources of error make Eq. 12.3.5 differ from the the true policy gradient in Eq. 12.3.4:

1. The *derivative* in $\frac{d\hat{h}_t}{d\theta}$ is based on an inaccurate model.
2. The *derivatives* in both $\nabla_{\hat{s}_t} r(\hat{s}_t)$ and $\frac{d\hat{h}_t}{d\theta}$ are evaluated along the wrong trajectory.

What ILC does is that although we cannot deal with the first source of error, we can at least run the system to get the actual trajectory instead of using the wrong one predicted by our approximate model. The resulting gradient is thus:

$$\nabla_\theta \hat{J}(\theta) = \sum_{t=0}^{T} \nabla_{s_t} r(s_t) \frac{d\hat{h}_t}{d\theta}\Big|_{s_0, s_1, \cdots s_{T-1}} \tag{12.3.6}$$

Brief Proof of Convergence and Optimality

It has been proved that if the model isn't too bad and the problem is well-behaved enough[3] then following the gradient will converge to a neighborhood of a local optimum. More formally:

$$\left\| \frac{df_t}{ds} - \frac{d\hat{f}_t}{ds} \right\|_2 \leq \epsilon \text{ and } \left\| \frac{df_t}{da} - \frac{d\hat{f}_t}{da} \right\|_2 \leq \epsilon \Rightarrow \left\| \nabla_\theta \hat{J} - \nabla_\theta J \right\|_2 \leq K\epsilon, \tag{12.3.7}$$

[3] Certain boundedness and smoothness conditions hold for the true MDP.

where K is a constant related to the properties of the problem, such as the dimensionality of the problem, upper bound for reward, horizon of the problem, etc.

Further, if an exact line search is done for each gradient ascent step (every gradient ascent step updates the parameter to the best parameter along the gradient direction), the algorithm converges to a region of local optimality,

$$\|\nabla_\theta J\|_2 \leq \sqrt{2}K\epsilon. \tag{12.3.8}$$

The above theorem guarantees that ILC converges to a region of local optimality. On the other hand, in practice when the policy is close the true optimal policy, it tends to oscillate without actually converging to the optimum.

12.4 Related Reading

[1] Abbeel, P., Quigley, M. and Ng, A.Y., *Using inaccurate models in reinforcement learning.*, ICML 2006.

,

13
Response Surface Methods

In this lecture we introduce Response Surface Methods (RSM) for policy optimization in reinforcement learning (RL). The problem setting in this lecture makes two important assumptions:

Parametric policy the policy $\pi : \mathbb{S} \to \mathbb{A}$ depends on a parameter vector $\theta \in \mathbb{R}^d$.

Episodic learning In this setting learning proceeds by interleaving two stages in a loop:

1. Batch simulation (or execution) of one or more "episodes"[1] under the current policy.
2. Adjustment of the policy (in this case, of the current policy's parameters) based on some aggregate performance metric from the simulations (e.g. total sum of rewards).

Episodic learning contrasts with methods that continuously learn throughout the simulation.

[1] Previously described as "rollouts" in this course.

See table 13.1 for an example of episodes, policies and rewards in three problem domains.

	Domain	
	Tetris	Helicopter
Episode	Game	Simulate for 1 minute
Policy	Board $\to$ Move	$(x, y, z, \dot{x}, \dot{y}, \dot{z}) \to$ (cycle, throttle, collective)
Reward	# of lines cleared	Remain close to desired trajectory

Table 13.1: Examples of episodes, policies and rewards

13.1 Optimization with Response Surface Methods

RSM is a general purpose, "black box" optimization method for any function $f : \mathbb{X} \to \mathbb{R}$. The outline of RSM is as follows:

1. Pick an initial point $\mathbf{x}_1$.
2. For $t = 1, \ldots, T$
   (a) Obtain response $f(\mathbf{x}_t)$.
   (b) Fit a "response surface" $\hat{f}$ to all data points $\{(\mathbf{x}_1, f(\mathbf{x}_1)), \ldots, (\mathbf{x}_t, f(\mathbf{x}_t))\}$.
   (c) Use the response surface to choose a new $\mathbf{x}_{t+1}$.

Note that t here means iterations rather than steps in episodes.

Step (b) involves fitting a function $\hat{f}$ and step (c) optimizes over the response surface $\hat{f}$. These steps may be computationally expensive. Hence, RSM is most useful when evaluating f is itself very costly compared to solving the surrogate problems in steps (b) and (c), which is often the case in reinforcement learning for robotics. If evaluating f is cheap, then we may be better off using one of black box methods covered earlier, such as the cross entropy method.

13.2 Fitting the response surface: Gaussian Process Regression

In step (b) of the algorithm we fit a response surface $\hat{f}$, sometimes called the *surrogate function*, to all our data points $\{(\mathbf{x}_1, f(\mathbf{x}_1)), \ldots, (\mathbf{x}_t, f(\mathbf{x}_t))\}$. Intuitively, $\hat{f}$ is an approximation the real f, but much less expensive to evaluate, and it will guide our search for the location of the optima in f.

Gaussian processes (GPs) are a commonly used model in response surfaces methods that maintain a nonparametric prior over functions. The intuition behind the GP prior is to see functions as a point in a continuous, "infinite-dimensional" space. The joint distribution for any finite set of samples $\{f(\mathbf{x}_1), \ldots, f(\mathbf{x}_t)\}$ is Gaussian and defined by a mean function $\mu : \mathbb{X} \to \mathbb{R}$ and a covariance function $k : \mathbb{X} \times \mathbb{X} \to \mathbb{R}$:

$$\begin{aligned} \big(f(\mathbf{x}_0), \ldots, f(\mathbf{x}_t)\big)^\top &\sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \\ \boldsymbol{\mu} &= \big(\mu(\mathbf{x}_1), \ldots, \mu(\mathbf{x}_t)\big)^\top \\ \boldsymbol{\Sigma}_{ij} &= k(\mathbf{x}_i, \mathbf{x}_j) \quad \forall\, (i,j) \in [1, \ldots, t]^2 \end{aligned}$$

Without loss of generality, the mean function μ is assumed to be always 0. This is the same as preprocessing the data as $f'(\mathbf{x}) = f(\mathbf{x}) - \mu(\mathbf{x})$.

Conceptually, the kernel function is like an "infinite-dimensional" covariance matrix where $\Sigma_{ij} = k(\mathbf{x}_i, \mathbf{x}_j) = \mathrm{cov}\big(f(\mathbf{x}_i), f(\mathbf{x}_j)\big)$. Therefore it must meet the equivalent of the SPD condition:

1. $k(\mathbf{x}_i, \mathbf{x}_j) = k(\mathbf{x}_j, \mathbf{x}_i), \quad \forall\, \mathbf{x}_i, \mathbf{x}_j$
2. Any matrix K s.t. $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$, that is symmetric positive semi-definite.

While we do not go into further detail here, we should note that choosing a suitable kernel function is very important for the performance of the GP regression and by extension, the RSM. This is a model selection problem, which falls outside the scope of this lecture.

GP inference

Given a prior for $f \sim GP(\mu, k)$ and a set of samples $\big((\mathbf{x}_1, f(\mathbf{x}_1)), \ldots, (\mathbf{x}_t, f(\mathbf{x}_t))\big)$, we wish to compute a posterior over $f(\mathbf{x}^*)$ at any query $\mathbf{x}^*$. We have

$$\begin{bmatrix} f(\mathbf{x}^*) \\ f(\mathbf{x}_1) \\ \vdots \\ f(\mathbf{x}_t) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} k(\mathbf{x}^*, \mathbf{x}^*) & k(\mathbf{x}^*, \mathbf{x}_1) & \ldots & k(\mathbf{x}^*, \mathbf{x}_t) \\ k(\mathbf{x}_1, \mathbf{x}^*) & k(\mathbf{x}_1, \mathbf{x}_1) & \ldots & k(\mathbf{x}_1, \mathbf{x}_t) \\ \vdots & \vdots & \ddots & \vdots \\ k(\mathbf{x}_t, \mathbf{x}^*) & k(\mathbf{x}_t, \mathbf{x}_1) & \ldots & k(\mathbf{x}_t, \mathbf{x}_t) \end{bmatrix} \right)$$

For notational convenience, let us define

$$\begin{aligned} K_{**} &= k(\mathbf{x}^*, \mathbf{x}^*) \\ K_{x*} &= \big[k(\mathbf{x}^*, \mathbf{x}_1), \ldots, k(\mathbf{x}^*, \mathbf{x}_t)\big]^\top \\ K_{xx} &= \big[k(\mathbf{x}_i, \mathbf{x}_j)\big]_{i,j=1}^N \end{aligned}$$

Using the properties of the Normal distribution, it can be shown that the posterior of $f(\mathbf{x}^*)$ is also a Gaussian:

$$f(\mathbf{x}^*) | f(\mathbf{x}_1), \ldots, f(\mathbf{x}_t) \sim \mathcal{N}\big(\mu_t(\mathbf{x}^*), \sigma_t(\mathbf{x}^*)\big) \tag{13.2.1}$$

$$\mu_t(\mathbf{x}^*) \doteq K_{x*}^\top K_{xx}^{-1} \big[f(\mathbf{x}_1), f(\mathbf{x}_1), \ldots f(\mathbf{x}_t)\big]^\top \tag{13.2.2}$$

$$\sigma_t(\mathbf{x}^*) \doteq K_{**} - K_{x*}^\top K_{xx}^{-1} K_{x*} \tag{13.2.3}$$

This gives us the posterior expected mean of $f(\mathbf{x})$ at any point $\mathbf{x}$ in the input space $\mathbb{X}$. It also gives us a posterior variance for $f(\mathbf{x})$, which we can interpret as a measure of uncertainty about the value of f at the point. In section 13.3 we will see how to use both in optimization.

Visualization

A good way to visualize a GP is to draw samples and plot them as functions. For this one selects a set of t samples $\mathbf{x}_i$, $i = 1, \ldots, t$, constructs the corresponding covariance matrix with k, samples n-dimensional points from the Gaussian with this covariance matrix, and plots them as functions. Figure 13.2.1 shows draws from two different GP priors.

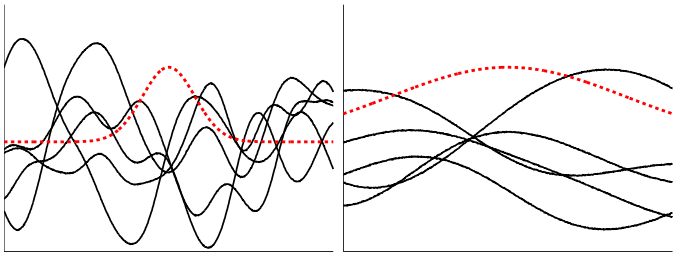


Figure 13.2.1: Draws from two different Gaussian Process Priors. Both use a squared exponential kernel $k(\mathbf{x}, \mathbf{x}') = \exp\{-(|\mathbf{x} - \mathbf{x}'|^2 / l^2)\}$, but with different characteristic length-scale l. The left figure has smaller l while the right figure has larger l. The kernel is graphically depicted by the red dotted line.

The same procedure can be applied to the posterior function, but using the posterior mean and covariance from eq. (13.2.1).

To learn more

This has been a very brief introduction to Gaussian Process regression. A good place to learn more is [1].

13.3 Choosing the next point to evaluate

For this section, we will switch notation to match the RL scenario:

- $f \to J$
- $\mathbf{x} \to \theta$

$J(\theta)$ is the "true cost-to-go" function, which is unknown but what we're trying to optimize as function of θ, the parameter vector for the policy π_θ. To evaluate $J(\theta)$ means to perform one (or more) episodes of simulation using the policy defined by θ.

Now, given all the samples $\big((\theta_1, J(\theta_1)), \dots (\theta_t, J(\theta_t))\big)$ observed so far and and a GP prior, we can estimate the posterior $J(\theta)$ for any query θ. Let us define

$$\begin{aligned} J(\theta) &\sim \mathcal{N}(\mu_t(\theta), \sigma_t(\theta)) \\ \mu_t(\theta) &\doteq \text{posterior mean for } J(\theta) \text{ according to eq. (13.2.2)} \\ \sigma_t(\theta) &\doteq \text{posterior variance of } J(\theta) \text{ according to eq. (13.2.3)} \end{aligned}$$

How do we choose the next point θ_{t+1} to evaluate (step (c) in the RSM algorithm outline)? Below we list some possible strategies, each with different advantages and disadvantages.

Maximum of posterior mean

In this strategy, we simply use

$$\theta_{t+1} = \operatorname*{argmax}_\theta \mu_t(\theta)$$

To obtain the argmax we can use any optimization algorithm, such as gradient descent, exhaustive search, Nelder-Mead, etc. (Note that this implies we are performing a nonlinear optimization step for each iteration of RSM, although on our estimated surrogate function, which is one of the reasons it is relatively slow).

This is the most "greedy" strategy, maximizing exploitation over exploration. As such it is more prone to converge in local optima. It may even choose $\theta_{t+1} = \theta_t$, in which case we will get stuck. Moreover, it ignores the uncertainty of the estimate. Nonetheless, it may be a good choice if we believe θ_t is close to a good optima.

Figure 13.3.1 shows this strategy in action. The first panel shows the true objective function $J(\theta)$, in black, and three evaluations (black dots). The rest are seven iterations of the RSM algorithm, ordered from left to right and top to bottom. The solid red line is the expected posterior mean of $J(\theta)$, $\mu_t(\theta)$. The dotted red lines are a confidence interval for $J(\theta)$, reflecting the magnitude of $\sigma_t(\theta)$. The blue line is the normalized value of the function we maximize to obtain θ_{t+1}, in this case μ_t. As we can see, in this case the strategy does nothing of interest, as the θ_{t+1} (the small black dot) equals one of the previously sampled points, and no further exploration occurs; the maximum is not found.

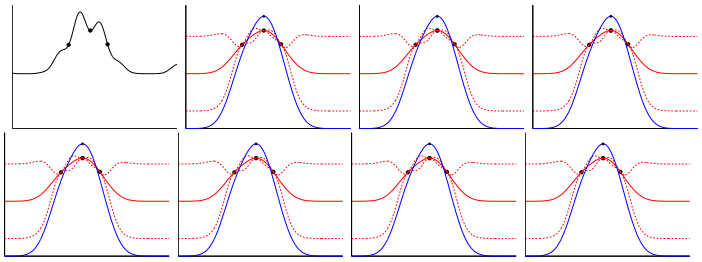


Figure 13.3.1: The "maximum of posterior mean" strategy in action. See text for explanation.

Maximum of posterior variance

In this strategy, we use

$$\theta_{t+1} = \operatorname*{argmax}_{\theta} \sigma_t(\theta)$$

That is, θ_{t+1} is placed where our uncertainty about $J(\theta)$ is greatest, regardless of the expected value. This is the "opposite" of the last strategy in the exploitation-exploration spectrum.

Figure 13.3.2 shows this strategy. We can see this strategy is good for exploring over various different locations, but fails to find the function maximum.

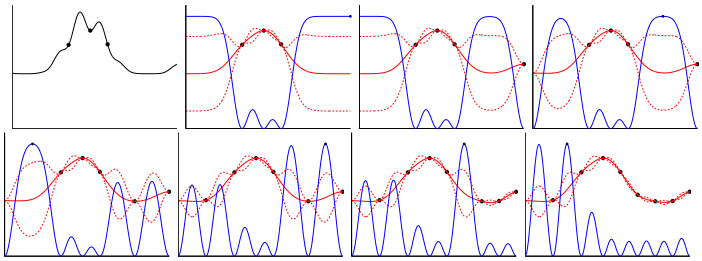


Figure 13.3.2: The "maximum of posterior variance" strategy in action. See text for explanation.

Maximum upper confidence bound

In practice, we want to balance exploitation and exploration. Therefore, a sensible strategy is to choose

$$\theta_{t+1} = \operatorname*{argmax}_{\theta} \mu_t(\theta) + \beta\, \sigma_t(\theta)$$

where β is a parameter regulating the exploitation-exploration tradeoff. $\mu_t(\theta) + \beta\, \sigma_t(\theta)$ can be interpreted as an "upper confidence bound" for $\mu_t(\theta)$. This strategy works well, but has the disadvantage of requiring a tuning parameter β.

Maximum probability of improvement

What we really want is to improve $J_{\max}$, the best value found so far.

Let us define an "improvement" function

$$I(\theta) = \max\big(J(\theta) - J_{\max}, 0\big)$$

Since $I(\cdot)$ depends on $J(\theta)$, it is also a random variable. One strategy is to maximize the *probability of improvement*:

$$\theta_{t+1} = \operatorname*{argmax}_{\theta} \Pr\big(I(\theta) > 0\big)$$

or alternatively,

$$\theta_{t+1} = \operatorname*{argmax}_{\theta} \int_{J_{\max}}^{\infty} \mathcal{N}\big(y|\mu_t(\theta), \sigma_t(\theta)\big)\mathrm{d}y$$

The main problem with this strategy is that if we ever choose $\theta_{t+1} = \theta_t$, we we will get stuck there. To avoid this we can use a modified improvement function

$$I(\theta_t) = \max\big(J(\theta_t) - (J_{\max} + \beta), 0\big)$$

But again, we have a tuning parameter β.

Maximum expected improvement

One problem with the last strategy is that we are only considering the probability of improvement, not the magnitude of the improvement. We can fix this by using the expected improvement:

$$\begin{aligned}\theta_{t+1} &= \operatorname*{argmax}_{\theta} \mathbb{E}\left[I(\theta)\right] \\ &= \operatorname*{argmax}_{\theta} \int_{J_{\max}}^{\infty} \mathcal{N}\big(y|\mu_t(\theta), \sigma_t(\theta)\big)\,\big(y - J_{\max}\big)\mathrm{d}y\end{aligned}$$

This has no tuning parameter and is one of the most popular RSM methods. One disadvantage is that it sometime tends to explore too much. Figure 13.3.3 shows this strategy in action. As we can see it has a good balance of exploration and exploitation, and finds the maximum.

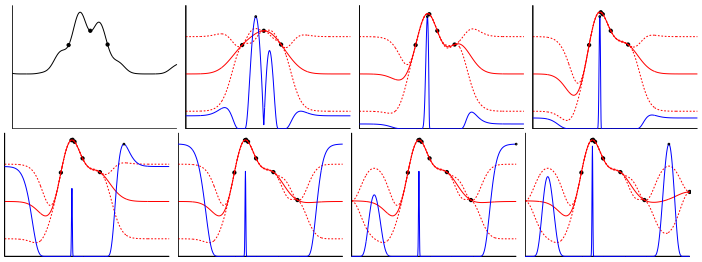


Figure 13.3.3: The "maximum of expected improvement" strategy in action. See text for explanation.

Miscellaneous

- For any strategy, at each step we can sample a random point with some probability ϵ. This encourages exploration.
- How do we take into account the stochasticity of $J(\theta)$ itself? Often it is simply ignored, which conflates it with our uncertainty about the value $J(\theta)$. Alternatively, it can be separately modelled as "process noise", which may be heteroscedastic (it varies with θ).
- We may take $J_{\max}$ to be stochastic. In this case the expected improvement requires a joint expectation calculation. Most people simply use the mean of the GP at the maximum point. Empirically it doesn't seem to matter too much.

13.4 Related Reading

[1] Carl Edward Rasmussen and Christopher K. I. Williams, *Gaussian Processes for Machine Learning*. MIT Press, 2006. www.gausianprocess.org

14
Comparing Black and Grey Box RL algorithms

15
Causality, Counterfactuals and Covariate Shift

16
Bibliography

P. Abbeel and A. Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the 21st International Conference on Machine Learning (ICML)*, 2004.

A. Agarwal, O. Chapelle, M. Dudík, and J. Langford. A reliable effective terascale linear learning system. *Journal of Machine Learning Research*, 15: 1111–1133, 2014. URL http://jmlr.org/papers/v15/agarwal14a.html.

Brian D. O. Anderson and John B. Moore. *Optimal Control: Linear Quadratic Methods*. Prentice-Hall, Inc., 1990.

B. D. Argall, S. Chernova, M. Veloso, and B. Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 2009.

S. Arora, E. Hazan, , and S. Kale. The multiplicative weights update method: A meta-algorithm and applications. *Theory of Computing*, 2012.

C. G. Atkeson. Using local trajectory optimizers to speed up global optimization in dynamic programming. In *Advances in Neural Information Processing Systems (NIPS)*, 1994.

C. G. Atkeson and J. Morimoto. Nonparametric representation of policies and value functions: A trajectory-based approach. In *Advances in Neural Information Processing Systems (NIPS)*, 2003.

C. G. Atkeson and S. Schaal. Robot learning from demonstration. In *Proceedings of the 14th International Conference on Machine Learning (ICML)*, 1997.

A. Bachrach, R. He, and N. Roy. Autonomous flight in unknown indoor environments. *International Journal of Micro Air Vehicles*, 1(4):217–228, 2009.

J. A. Bagnell. Robust supervised learning. In *Proceedings of the American Association for Artificial Intelligence (AAAI)*. AAAI Press / The MIT Press, 2005.

J. A. Bagnell and J. Schneider. Covariant policy search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

J. A. Bagnell, A. Y. Ng, S. Kakade, and J. Schneider. Policy search by dynamic programming. In *Advances in Neural Information Processing Systems*, 2003.

L. C. Baird. Residual algorithms: Reinforcement learning with function approximation. In *International Conference on Machine Learning*, 1995.

C. L Baker, R. Saxe, and J. B Tenenbaum. Action understanding as inverse planning. *Cognition*, 113(3):329–349, 2009.

Y. Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2009.

Justin A Boyan and Andrew W Moore. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in neural information processing systems*, 1995.

S. P Boyd, L. El Ghaoui, E. Feron, and V. Balakrishnan. *Linear Matrix inequalities in system and control theory*, volume 15. SIAM, 1994.

Massimo Caccia, Lucas Caccia, William Fedus, Hugo Larochelle, Joelle Pineau, and Laurent Charlin. Language gans falling short. *CoRR*, abs/1811.02549, 2018.

N. Cesa-Bianchi, Y. Freund, D. Haussler, D. P. Helmbold, R. E. Schapire, and M. K. Warmuth. How to use expert advice. *Journal of the ACM*, 1997.

A. Coates, P. Abbeel, and A. Y Ng. Apprenticeship learning for helicopter control. *Communications of the ACM*, 52(7):97–105, 2009.

H. Daumé III, J. Langford, and D. Marcu. Search-based structured prediction. *Machine Learning*, 2009.

A. Dragan and S. Srinivasa. Formalizing assistive teleoperation. In *Robotics: Science and Systems*, 2012.

Miroslav Dudik, Steven J. Phillips, and Robert E. Schapire. Performance guarantees for regularized maximum entropy density estimation. volume 3120, pages 472–486, 2004.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Peter D. Grunwald and A. Philip Dawid. Game theory, maximum entropy, minimum discrepancy and robust bayesian decision theory, 2004. URL http://arxiv.org/abs/math/0410076.

X. Guo, S. Singh, H. Lee, R. L Lewis, and X. Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in Neural Information Processing Systems*, 2014.

H. He, H. Daume III, and J. Eisner. Imitation learning by coaching. In *NIPS*, 2012.

Martial H Hebert. *Intelligent unmanned ground vehicles: autonomous navigation research at Carnegie Mellon*. Kluwer Academic Publishers, 1997.

Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. 2016. URL http://arxiv.org/abs/1606.03476.

D.A. Huang, A.M. Farahmand, K. M. Kitani, and J. A. Bagnell. Approximate maxent inverse optimal control and its application for mental simulation of human interactions. In *AAAI Conference on Artificial Intelligence*, 2015.

L. D Jackel, E. Krotkov, M. Perschbacher, J. Pippine, and C. Sullivan. The DARPA LAGR program: Goals, challenges, methodology, and phase i results. *Journal of Field Robotics*, 23(11-12):945–973, 2006.

D. H. Jacobson and D. Q. Mayne. *Differential Dynamic Programming*. Elsevier, 1970.

Arthur Jacot-Guillarmod, Franck Gabriel, and Clement Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 8580–8589. Curran Associates, Inc., 2018. URL http://papers.nips.cc/paper/8076-neural-tangent-kernel-convergence-and-generalization-in-neural-networks.pdf.

E. T. Jaynes. *Probability theory: The logic of science*. Cambridge University Press, 2003.

S. Kakade and J. Langford. Approximately optimal approximate reinforcement learning. In *Proceedings of the 19th International Conference on Machine Learning (ICML)*, 2002.

M. Kalakrishnan, J. Buchli, P. Pastor, M. Mistry, and S. Schaal. Learning, planning, and control for quadruped locomotion over challenging terrain. *The International Journal of Robotics Research*, 30(2):236–258, 2011.

R. E. Kalman. When is a linear control system optimal? *Journal of Fluids Engineering*, 86(1):51–60, 1964.

B. Kim, A. M. Farahmand, J. Pineau, and D. Precup. Learning from limited demonstrations. In *Advances in Neural Information Processing Systems*, pages 2859–2867, 2013.

K. M Kitani, B. D. Ziebart, J. A. Bagnell, and M. Hebert. Activity forecasting. In *European Conference on Computer Vision*. Springer, October 2012.

J. Kober and J. Peters. Imitation and reinforcement learning - practical algorithms for motor primitive learning in robotics. 2010.

J Zico Kolter, Pieter Abbeel, and Andrew Y Ng. Hierarchical apprenticeship learning with application to quadruped locomotion. In *Advances in Neural Information Processing Systems*, pages 769–776, 2007.

H. Kretzschmar, M. Kuderer, and W. Burgard. Learning to predict trajectories of cooperatively navigating agents. In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, Hong Kong, China, 2014. DOI: 10.1109/ICRA.2014.6907442. URL http://ais.informatik.uni-freiburg.de/publications/papers/kretzschmar14icra.pdf.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *IEEE*, 1998.

J. Leonard, J. How, S. Teller, M. Berger, S. Campbell, G. Fiore, L. Fletcher, E. Frazzoli, A. Huang, S. Karaman, et al. A perception-driven autonomous urban vehicle. *Journal of Field Robotics*, 25(10):727–774, 2008.

S. Levine and V. Koltun. Continuous inverse optimal control with locally optimal examples. In *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, pages 41–48, 2012.

Lihong Li. A worst-case comparison between temporal difference and residual gradient with linear function approximation. *Proceedings of the 25th international conference on Machine learning - ICML '08*, pages 560–567, 2008. DOI: 10.1145/1390156.1390227. URL http://portal.acm.org/citation.cfm?doid=1390156.1390227.

L. Ljung. Convergence analysis of parametric identification methods. *IEEE Transactions on Automatic Control*, 1978.

O. Miksik, D. Munoz, J. A. Bagnell, and M. Hebert. Efficient temporal consistency for streaming video scene analysis. In *ICRA*, 2013.

K. Mülling, J. Kober, O. Kroemer, and J. Peters. Learning to select and generalize striking movements in robot table tennis. *The International Journal of Robotics Research*, 32(3):263–279, 2013.

D. Munoz, J. A. Bagnell, and M. Hebert. Stacked hierarchical labeling. In *ECCV*, 2010.

M. Nechyba and J. A. Bagnell. Stabilizing human control strategies through reinforcement learning. In *Proc. IEEE Hong Kong Symp. on Robotics and Control*, volume 1, pages 39–44, April 1999.

A. Y Ng and S. J Russell. Algorithms for inverse reinforcement learning. In *Icml*, pages 663–670, 2000.

Takayuki Osa, Joni Pajarinen, and Gerhard Neumann. *An Algorithmic Perspective on Imitation Learning*. Now Publishers Inc., Hanover, MA, USA, 2018. ISBN 168083410X, 9781680834109.

D. Pomerleau. ALVINN: An Autonomous Land Vehicle in a Neural Network. In *Advances in Neural Information Processing Systems (NIPS)*, 1989.

M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.

V. Ramakrishna, D. Munoz, M. Hebert, J. A. (Drew) Bagnell, and Y. A. Sheikh. Pose machines: Articulated pose estimation via inference machines. In *European Conference on Computer Vision*, 2014.

N. Ratliff. *Learning to Search: Structured Prediction Techniques for Imitation Learning*. PhD thesis, Carnegie Mellon University, 2009.

N. Ratliff, J. A. Bagnell, and M. Zinkevich. (Online) subgradient methods for structured prediction. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2007.

N. Ratliff, B. Ziebart, K. Peterson, J A. Bagnell, M. Hebert, A. K. Dey, and S. Srinivasa. Inverse optimal heuristic control for imitation learning. AISTATS, 2009a.

N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa. Chomp: Gradient optimization techniques for efficient motion planning. In *ICRA*, 2009b.

Nathan D Ratliff, J Andrew Bagnell, and Martin A Zinkevich. Maximum margin planning. In *Proceedings of the 23rd international conference on Machine learning*, pages 729–736, 2006.

Nathan D Ratliff, David Silver, and J Andrew Bagnell. Learning to search: Functional gradient techniques for imitation learning. *Autonomous Robots*, 27(1):25–53, 2009c.

S. Ross. Comparison of imitation learning approaches on Super Tux Kart, 2010a. URL http://www.youtube.com/watch?v=V00npNnWzSU.

S. Ross. Comparison of imitation learning approaches on Super Mario Bros, 2010b. URL http://www.youtube.com/watch?v=anOI0xZ3kGM.

S. Ross and J. A. Bagnell. Efficient reductions for imitation learning. In *AISTATS*, 2010.

S. Ross and J. A. Bagnell. Reinforcement and imitation learning via interactive no-regret learning. *arXiv preprint arXiv:1406.5979*, 2014.

S. Ross, G. J. Gordon, and J. A. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS*, 2011a.

S. Ross, D. Munoz, M. Hebert, and J. A. Bagnell. Learning message-passing inference machines for structured prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2011b.

S. Ross, N. Melik-Barkhudarov, K. S. Shankar, A. Wendel, D. Dey, J. A. Bagnell, and M. Hebert. Learning monocular reactive uav control in cluttered natural environments. In *ICRA*, 2013a.

S. Ross, N. Melik-Barkhudarov, K. S. Shankar, A. Wendel, D. Dey, J. A. Bagnell, and M. Hebert. Autonomous vision-based flight through a forest, 2013b. URL http://www.youtube.com/watch?v=oO_Ohp1pHBw.

S. Ross, J. Zhou, Y. Yue, D. Dey, and J. A. Bagnell. Learning policies for contextual submodular prediction. In *ICML*, 2013c.

Stephane Ross and J. Andrew Bagnell. Stability Conditions for Online Learnability. *arXiv:1108.3154*, 2011. URL http://arxiv.org/abs/1108.3154.

J. Rust. *Do people behave according to Bellman's principle of optimality?* Hoover Institution, Stanford University, 1992.

John Rust. Structural estimation of markov decision processes. *Handbook of econometrics*, 4(4), 1994.

A. Safonova and J. K Hodgins. Construction and optimal search of interpolated motion graphs. In *ACM Transactions on Graphics (TOG)*, volume 26, page 106. ACM, 2007.

Ankan Saha, Prateek Jain, and Ambuj Tewari. The Interplay Between Stability and Regret in Online Learning. *arXiv preprint arXiv:1211.6158*, pages 1–19, 2012. URL http://arxiv.org/abs/1211.6158.

Robert E. Schapire and Manfred K. Warmuth. On the worst-case analysis of temporal-difference learning algorithms. *Machine Learning*, 22(1):95–121, 1996. ISSN 0885-6125. DOI: 10.1007/BF00114725.

S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

D. Silver. *Learning Preference Models for Autonomous Mobile Robots in Complex Domains*. PhD thesis, Carnegie Mellon University, 2010.

Bharath K. Sriperumbudur, Arthur Gretton, Kenji Fukumizu, Gert R. G. Lanckriet, and Bernhard Schölkopf. A note on integral probability metrics and ϕ-divergences. 2009. URL http://arxiv.org/abs/0901.2698.

Wen Sun and J Andrew Bagnell. Online bellman residual algorithms with predictive error guarantees. *UAI*, 2015.

Wen Sun and J. Andrew (Drew) Bagnell. Online bellman residual and temporal difference algorithms with predictive error guarantees. In *Proceedings of The 25th International Joint Conference on Artificial Intelligence - IJCAI 2016*, April 2016.

Wen Sun, Geoffrey J Gordon, Byron Boots, and J Bagnell. Dual policy iteration. In *Advances in Neural Information Processing Systems*, 2018.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

U. Syed and R. E Schapire. A game-theoretic approach to apprenticeship learning. In *Advances in neural information processing systems*, pages 1449–1456, 2007.

Yuval Tassa, Tom Erez, and Emanuel Todorov. Synthesis and stabilization of complex behaviors through online trajectory optimization. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 4906–4913. IEEE, 2012.

C. Urmson, J. Anhalt, J. A. Bagnell, C. Baker, R. Bittner, MN Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, et al. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8): 425–466, 2008.

P. Vernaza and D. D. Lee. Efficient dynamic programming for high-dimensional, optimal motion planning by spectral learning of approximate value function symmetries. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2011.

K. Waugh, B. D. Ziebart, and J. A. Bagnell. Computational rationalization: The inverse equilibrium problem. In *Proceedings of the International Conference on Machine Learning*, June 2011.

Shih-En Wei, Varun Ramakrishna, Takeo Kanade, and Yaser Sheikh. Convolutional pose machines. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4724–4732, 2016.

C. Wellington and A. Stentz. Online adaptive rough-terrain navigation vegetation. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, volume 1, pages 96–101 Vol.1, 2004.

B. D. Ziebart, A. Maas, J. A. Bagnell, and A. K. Dey. Maximum entropy inverse reinforcement learning. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI)*, 2008a.

B. D. Ziebart, A. Maas, A. Dey, and J. A. Bagnell. Navigate like a cabbie: Probabilistic reasoning from observed context-aware behavior. In *UBICOMP: Ubiquitious Computation*, 2008b.

B. D. Ziebart, N. Ratliff, G. Gallagher, C. Mertz, K. Peterson, J. A. Bagnell, M. Hebert, A. Dey, and s. Srinivasa. Planning-based prediction for pedestrians. In *Proc. IROS 2009*, October 2009.

B. D. Ziebart, J. Andrew Bagnell, and A. K. Dey. Modeling interaction via the principle of maximum causal entropy. In *Proceedings of the 27th International Conference on Machine Learning*, 2010.

B. D. Ziebart, A. Dey, and J. A. Bagnell. Probabilistic pointing target prediction via inverse optimal control. In *International Conference on Intelligent User Interfaces (IUI 2012)*, 2012.

B. D. Ziebart, J. A. Bagnell, and A. K. Dey. The principle of maximum causal entropy for estimating interacting processes. *IEEE Transactions on Information Theory*, February 2013.

M. Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *Proceedings of the 20th International Conference on Machine Learning (ICML)*, 2003.

M. Zucker, N. Ratliff, M. Stolle, J. Chestnutt, J Andrew Bagnell, C. G. Atkeson, and J. Kuffner. Optimization and learning for rough terrain legged locomotion. *The International Journal of Robotics Research*, 30(2):175–191, 2011.