046194 - Learning and Planning in Dynamical Systems

Edited by Aviv Tamar and based on notes by Shie Mannor and Nahum Shimkin

Spring 2019

Contents

Chapter 1

Introduction and Overview

1.1 Sequential Decision Problems

The focus of this course is the solution of *sequential decision problems*. In such problems, a sequence of decisions is required, where each one has partial effect on the outcome. Sequential decision problems are common in a wide variety of application domains, that include:

- Artificial intelligence: Robotic motion planning, mission planning, board games (chess, backgammon), computer games (game of strategy, football).
- Operations research: Inventory control, supply chain management, project assignment, scheduling of servers and customers.
- Finance: Investment strategies, financial planning, gambling.
- Communication networks and computer systems: Packet routing, flow control, cache handling, job scheduling, power control.
- Control Engineering: High-level feedback control of dynamic systems such as land/sea/air vehicles, chemical processes, power systems, analog amplifiers, etc.

We will examine two basic challenges in the context of these decision problems:

1. **Planning**: Optimal selection of the required sequence of actions (called an optimal control policy) to minimize a specified cost function in a given system model, when the model is *known in advance*.
2. **Learning**: When a system model is not available for the planner, it may need to be learned during operation, by trying out different options and observing their consequences.

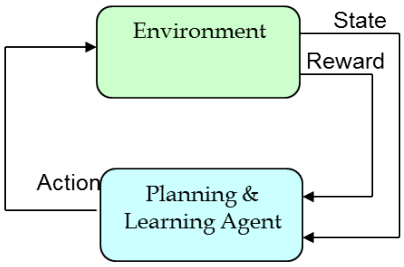


Figure 1.1: The situated agent

Our main *planning* tools will rely on the theory of *Dynamic Programming* (DP), which offers a set of methods and algorithms for solving sequential decision problems.

Reinforcement Learning (RL) is the area of machine learning that deals with learning by observing the consequences of different actions. The term, which is borrowed from the behaviorist school in psychology, hints to the use of positive reinforcement to encourage certain behaviors. It is now used in machine learning to refer generally to learning in sequential decision processes and dynamical systems.

The situated agent viewpoint: A basic viewpoint that underlies the field of RL, especially within AI, is that of the situated agent. The agent here may be a cognitive entity or a computer algorithm. It interacts with an external environment (the controlled system) by observing the environment state, and taking actions. Once an action is taken a reward signal is obtained, the system moves to a new state, and the cycle is repeated. Designing an effective generic learning agent in such a framework may be seen as an ultimate goal in AI, and is a major driver of RL research.

Example 1.1. ***The Tetris player***

Consider designing an AI player for the game of Tetris. The environment here would be the game simulator, and the environment state at a particular time corresponds to the game board at that time, and the shape of the new piece that just arrived. Upon observing a system state, the agent (the AI player) takes an action - it decides where on the board to position the new piece. Consequentially, if rows on the board were cleared, the agent receives a corresponding reward, and the the next system state is determined by the simulator.

Markov Decision Processes (MDPs) are the standard model that allows to treat these planning and learning problems in a unified manner. An MDP describes a sequential decision problem, with the following properties:

- A controlled dynamical system, with state-based dynamics. At each state some decision (or action) needs to be taken, and as a consequence the system moves to the next state.

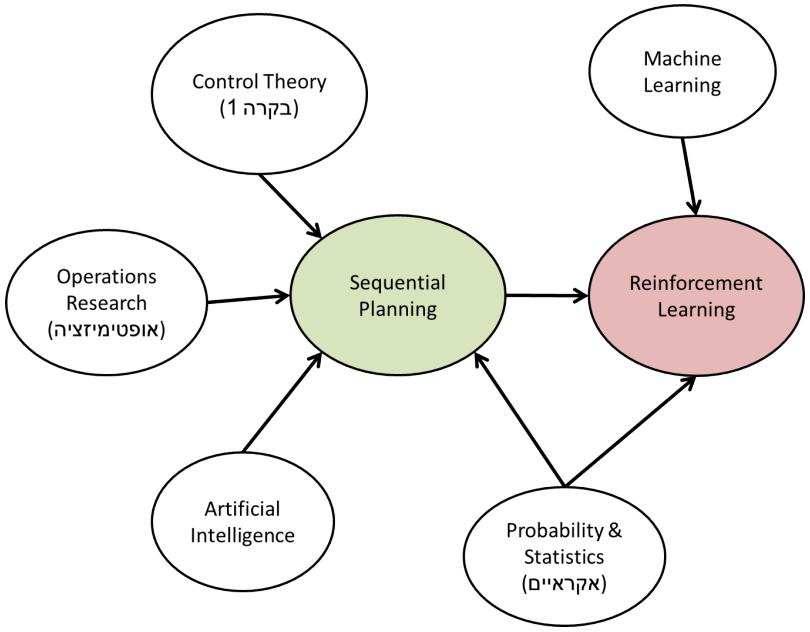
- A reward function is associated with state-action pair, and the system performance is measured by the accumulated rewards.
- MDPs are especially suited to handle discrete problems (discrete state space, discrete action set, discrete time).
- MDPs allow to model in a natural way stochastic effects, and in particular stochastic system dynamics. We note that MDPs are currently the standard model for probabilistic planning in AI.

Challenges - what makes the planning problem hard?

- Lack of a global structure - most systems of interest do not have a simple global structure (such as that of a linear system). Therefore, planning needs to address each state (or group of states) in a distinct way.
- Large state-spaces: Many problems of interest have a huge number of states (recall the tetris example - how many board configurations are possible?), which makes exact solution impossible, even with the strongest algorithms. *We note that models with a large number states are often the result of combinatorial explosion in state vectors with a large number of components. This phenomenon is commonly known as the curse of dimensionality.*
- Large action-spaces: Some problems, especially in resource allocation, also exhibit large action-spaces, which present additional computational challenges.
- Stochastic effects - these are successfully handled within the MDP model, although they may add to the computational challenge.
- Uncertain or unknown environments – which leads to the application of Reinforcement Learning techniques.

The computational challenge of complex problems (in particular, large state spaces) requires the use of approximation techniques for planning. This is an active research field, which is called Approximate Dynamic Programming (ADP) in our context. It is interesting to note that Reinforcement Learning techniques are often used as a means for planning in complex systems, even if the system dynamics is known.

The following diagram illustrates the relation and influence between the different disciplines involved.



1.2 Some Illustrative Planning Examples

We next provide several examples that serve to illustrate the models involved. We start with planning examples.

1.2.1 Shortest Path on a Graph

Finding a shortest path between a source and destination nodes (or vertices) on a weighted graph is a generic problem with numerous applications, from routing in communication networks to trajectory planning, puzzle solving, and general AI planning. Essentially, Dynamic Programming based algorithms compute the distance from every node in the graph to the destination, spreading out from the destination until the source is reached. Specific algorithms for this problem include the Bellman-Ford algorithm, and Dijkstra's algorithm and its variants, such as A*. This problem is widely discussed in courses on AI, and we will only briefly touch upon the main algorithms. We will also consider its stochastic generalization, the Stochastic Shortest Path problem.

1.2.2 The Secretary Problem

Problem setup: We are interviewing n candidates for a job. We know that the candidates are strictly ordered in quality from 1 to n, but interview them in random order. Once we interview a candidate, his or her quality can be evaluated *relative* to previously

interviewed candidates. A candidate that is not hired during the interview cannot be recalled.

The planning problem: Our goal is to find a rule for choosing a candidate, that will maximize the chances of hiring the best one.

This problem and its generalizations have been widely studied in the operation research and computer science literature. It is an instance of a wider class of *optimal stopping problems*.

Exercise 1.1. Let $B_t \sim$ Bernoulli i.i.d. for all $t = 1, 2, 3, \ldots$. Consider the empirical average:

$$X_\tau = \frac{1}{\tau} \sum_{t=1}^{\tau} B_t.$$

where τ is a stopping time. The objective is to find a stopping rule for τ that maximizes $\mathbb{E}[X_\tau]$.

1. Find a stopping rule that achieves $\mathbb{E}[X_\tau] \geq 0.75$
2. (Hard!) What is the maximal $\mathbb{E}[X_\tau]$ that can be achieved?

1.2.3 Inventory Management

System description: Suppose we manage a single-item inventory system with daily orders. On the morning of each day k, we observe the current inventory x_k, and can order any quantity $a_k \geq 0$ of that item that arrives immediately. The daily demand for the item is denoted D_k. Thus, the number of items sold on each day k is $\min\{D_k, x_k + a_k\}$, and the inventory at the end of the day is $x_{k+1} = [x_k + a_k - D_k]^+$. In this model we can determine the order a_k, whereas the demand D_k is uncertain and is modeled as a random variable with known probability distribution. We assume that D_k is a sequence of *independent* random variables.

Cost structure: Suppose that the cost for an order of a_k is $J(a_k)$, a price P is charged for each sold item, and a penalty $C(x_k + a_k - D_k)$ is incurred for over or under demand miss (where $C(y) \geq 0$ and $C(0) = 0$). Therefore, the net return (or reward) for day k is

$$R_k = P \min\{D_k, x_k + a_k\} - J(a_k) - C(x_k + a_k - D_k).$$

The *cumulative return* is the sum of daily returns over a given period, say 1 to K.

The planning problem: For each $k = 1, \ldots, K$, we observe the current inventory x_k, and need to determine the amount a_k to order next. The goal is to maximize the *expected value* of the cumulative return over the given period:

$$E(\sum_{k=1}^{K} R_k) \ \rightarrow \ \min$$

Remark 1.1. *: A common policy in inventory management is the (s, S) replenishment policy: whenever the inventory x drops below s, order $S - x$ items. This policy can be shown to be optimal under appropriate assumptions on the cost.*

Exercise 1.2. For the single-stage problem ($K = 1$) with $R = 0$, $J \equiv 0$, $C(y) = y^2$, show that a (s, S) policy is optimal (with $s = S$).

1.2.4 Admission control to a queueing system

System description: Consider a single-server queue, to which jobs (or customers) arrive sequentially. Each job has a service demand (in time unit of service) that is represented by a random variable with known distribution. The arrival process is, say, Poisson with rate λ. The system manager can deny admission to an arriving job.

Cost structure: Suppose that the system incurs a penalty of C per unit time for each job waiting in the queue, and a reward R for each customer served. We wish to minimize the expected cumulative cost over a period of time.

The planning problem: Suppose a job arrives when the queue size is $x \geq 0$. Should it be admitted or denied entry? This problem represents a basic example of *queueing control problems*. A wide range of computer, communication, and service systems can be represented by networks of queues (waiting lines) and servers. Efficient operation of these systems gives rise to a wide range of dynamic optimization problems. These include, among others, job admission control, job routing, server and job scheduling, and control of service rates.

1.2.5 Stochastic scheduling

System description: Suppose we have a single server that caters to n classes of jobs, one job at a time. Here the server may be human, a router in a communication network, a CPU in a computer system, etc. Jobs of class i arrive as a Poisson process with rate λ_i, and each requires a service duration D_i (which could be random – for example, an exponentially-distributed random variable with expected value μ_i^{-1}).

Cost structure: Jobs of class i incur a waiting cost C_i per unit time of waiting for service, and a reward R_i for completing service.

The planning problem: Suppose that the server becomes available at an instance t, and sees the state vector $X(t) = (x_1(t), \ldots, x_n(t))$, where x_i is the number of class i jobs waiting for service. Which job class should the server attend next?

1.3 Some Illustrative Learning Examples

The following examples address the learning problem.

1.3.1 The Multi-Armed Bandit (MAB) problem

This is an important problem in statistics and machine learning. The exotic name derives from casino-like slot machines, which are also known as "single-armed bandits". Suppose we are faced with N different arms (slot machines), which may have different win probabilities (for simplicity assume 0-1 results, with obvious generalizations). These probabilities are not known to us. There are two different goals in this problem:

1. To identify the best arm (pure exploration).

2. To maximize our gains over a large time horizon (exploration vs. exploitation).

This model, under the second objective, presents in its purest form the **exploration vs. exploitation dilemma**, which is fundamental in Reinforcement Learning: While each arm may be optimal and should be sufficiently tried out, we should also focus at some point on the arm that seems to be the best. This tradeoff must be addressed by an appropriate *exploration strategy*.

We note that this problem can be considered as a special case of the MDP model, with a single state (i.e., static environment). Its original motivation was related to experimental drug (medicine) administration. Its current popularity in Machine Learning owes to recent on-line application such as ad posting: choosing the right class of ads to post in a given space or context to maximize revenue (or "click" count).

1.3.2 Learning to Play Chess / Backgammon

Current AI programs for chess playing essentially rely on extensive search in the game tree, starting from the current board position. That search is aided by an evaluation function (the *heuristic*, or *value function*), which provides a numerical value to each board position that estimates it strength (say, the likelihood of a win).

Suppose for simplicity that the opponent's playing strategy is known, so that the problem reduces to a single-player planning problem. The problem, of course, is huge number of states (board positions), which rules out an exact and complete solution. Therefore, partial solutions guided by heuristic and empirical rules must be used.

Machine learning offers a set of tools to improve the performance of a given AI player, by data-based tuning key parameter of the algorithm. Such improvements are often imperative

for achieving state-of-the-art performance. Specific topics in which RL techniques have been successfully applied include:

- Value function learning (by self-play and record analysis).
- Powerful heuristics for guiding the tree search.

These techniques have been applied in the games of Backgammon, Chess, and recently Go and Poker.

1.3.3 Skill Learning in Robotics

Suppose we wish to program a robot to juggle a ball on a racket, or to walk efficiently. We can start by programming the robot as best we can, but in most cases we will not be able to obtain fully efficient operation that way, and many "fine tunings" will be required. One of the major approaches for improvement is to equip the robot with learning capabilities, so that it can improve its performance over time. We can distinguish here between two learning goals:

a. Short term learning, or adaptivity - adapting to changing environment conditions (such as the weight of the ball, wind, etc.).

b. Skill learning - acquiring and improving the basic capability to perform certain motions or tasks in an efficient manner.

A major Reinforcement Learning approach to these learning problems is a set of methods called *direct policy search*. Here the required task or motion is parameterized by a vector of continuous parameters, which are tuned by simulation or during actual operation using gradient-based methods. We will touch upon this important topic towards the end of the course.

1.4 Mathematical Tools

We briefly mention the main mathematical tools that will be used throughout the course.

1. Concentration inequalities
2. MDPs - dynamic programming
3. Optimization - algorithms and theory
4. Generalization and risk (PAC)
5. Approximation theory
6. Convergence of stochastic processes

Chapter 2

Deterministic Decision Processes

In this chapter we introduce the dynamic system viewpoint of the optimal planning problem. We restrict the discussion here to deterministic (rather than stochastic) systems, and consider the finite-horizon decision problem and its recursive solution via finite-horizon Dynamic Programming.

2.1 Discrete Dynamic Systems

We consider a discrete-time dynamic system, of the form:

$$x_{k+1} = f_k(x_k, u_k), \quad k = 0, 1, 2, \ldots, N-1$$

where

- k is the time index.
- $x_k \in X_k$ is the state variable at time k, and X_k is the set of possible states at time k.
- $u_k \in U_k$ is the control variable at time k, and U_k is the set of possible control values (or actions) at time k.
- $f_k : X_k \times U_k \to X_{k+1}$ is the state transition function, which defines the *state dynamics* at time k.
- $N > 0$ is the *time horizon* of the system. It can be finite or infinite.

Remark 2.1. *More generally, the set U_k of available actions may depend on the state at time k, namely: $u_k \in U_k(x_k) \subset U_k$.*

Remark 2.2. *The system is in general time-varying. It is called* time invariant *if f_k, X_k, U_k do not depend on the time k. In that case we write*

$$x_{k+1} = f(x_k, u_k), \quad k = 0, 1, 2, \ldots, N-1; \quad x_k \in X,\ u_k \in U(x_k).$$

Remark 2.3. *The state dynamics may be augmented by an output equation:*

$$y_k = h_k(x_k, u_k)$$

where y_k is the system output, or the observation. In most of this course we implicitly assume that $y_k = x_k$, namely, the current state x_k is fully observed.

Example 2.1. ***Linear Systems***

The best known example of a dynamic system is that of a linear time-invariant system, where:

$$x_{k+1} = Ax_k + Bu_k$$

with $x_k \in \mathbb{R}^n$, $u_k \in \mathbb{R}^m$. Here the state and action spaces are evidently continuous (as opposed to discrete).

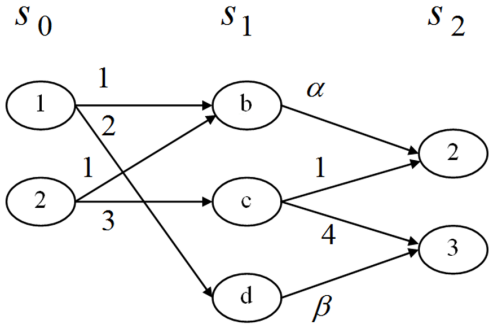
Example 2.2. ***Finite models***

Our emphasis here will be on finite state and action *models. A finite state space contains a finite number of points: $X_k = \{1, 2, \ldots, n_k\}$. Similarly, a finite action space implies a finite number of control values at each stage:*

$$U_k(x) = \{1, 2, \ldots, m_k(x)\}, \quad x \in X_k$$

Notation for finite models: When the state and action spaces are finite, it is common to denote the state by s_k (instead of x_k) and the actions by a_k (instead of u_k). That is, the system equations are written as: $s_{k+1} = f_k(s_k, a_k), \quad k = 0, 1, 2, \ldots, N-1$ with $s_k \in S_k$, $a_k \in A_k(x_k) \subset A_k$. **We will adhere to that notation in the following**.

Graphical description: Finite models (over finite time horizons) can be represented by a corresponding decision graph:

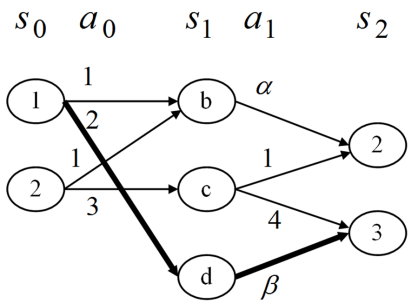


Here:

- $N = 2$, $S_0 = \{1, 2\}$, $S_1 = \{b, c.d\}$, $S_2 = \{2, 3\}$,

- $A_0(1) = \{1, 2\}$, $A_0(2) = \{1, 3\}$, $A_1(b) = \{\alpha\}$, $A_1(c) = \{1, 4\}$, $A_1(d) = \{\beta\}$
- $f_0(1, 1) = b$, $f_0(1, 2) = d$, $f_0(2, 1) = b$, $f_0(2, 3) = c$, $f_1(b, \alpha) = 2$, etc.

Definition 2.1. ***Feasible Path***
A feasible path for the specified system is a sequence $(s_0, a_0, \ldots, s_{N-1}, a_{N-1}, s_N)$ *of states and actions, such that* $s_k \in A_k(s_k)$ *and* $s_{k+1} = f_k(s_k, a_k)$.



A feasible path

2.2 The Finite Horizon Decision Problem

We proceed to define our first and simplest planning problem. For that we need to specify a *performance objective* for our model, and the notion of *control policies*.

2.2.1 Costs and Rewards

The cumulative cost: Let $h_N = (s_0, a_0, \ldots, s_{N-1}, a_{N-1}, s_N)$ denote an N-stage path for the system.. To each feasible path h_N we wish to assign some cost $C_N = C_N(h_N)$.

The standard definition of the cost C_N is through the following *cumulative cost functional*:

$$C_N(h_N) = \sum_{k=0}^{N-1} c_k(s_k, a_k) + c_N(s_N)$$

Here:

- $c_k(s_k, a_k)$ is the *instantaneous* cost or *single-stage* cost at stage k, and c_k is the instantaneous cost function.
- $c_N(s_N)$ is the *terminal* cost, and c_N is the terminal cost function.

Note:

- The cost functional defined above is *additive* in time. Other cost functionals are possible, for example the max cost, but additive cost is by far the most common and useful.
- We shall refer to C_N as the *cumulative N-stage cost*, or just the *cumulative cost*.

Our objective is to *minimize* the cumulative cost C_N, by a proper choice of actions. We will define that goal more formally below.

Cost versus reward formulation: It is often more natural to consider *maximizing* reward rather than minimizing cost. In that case, we define the cumulative N-stage return function:

$$R_N(h_N) = \sum_{k=0}^{N-1} r_k(s_k, a_k) + r_N(s_N)$$

Here and r_k is the running reward, and r_N is the terminal reward. Clearly, minimizing C_N is equivalent to maximizing R_N, if we set:

$$r_k(s, a) = -c_k(s, a) \text{ and } r_N(s) = -c_N(s).$$

2.2.2 Optimal Paths

Our first planning problem is the following *Path Optimization Problem*:

- For a given initial state s_0, find a feasible path $h_N = (s_0, a_0, \ldots, s_{N-1}, a_{N-1}, s_N)$ that minimizes the cost functional $C_N(h_N)$, over all feasible paths h_N.

Such a path is called an *optimal path* from s_0.

A more general notion than a path is that of a *control policy*, that specifies the action to be taken at each state. Control policies will play an important role in our Dynamic Programming algorithms, and are defined next.

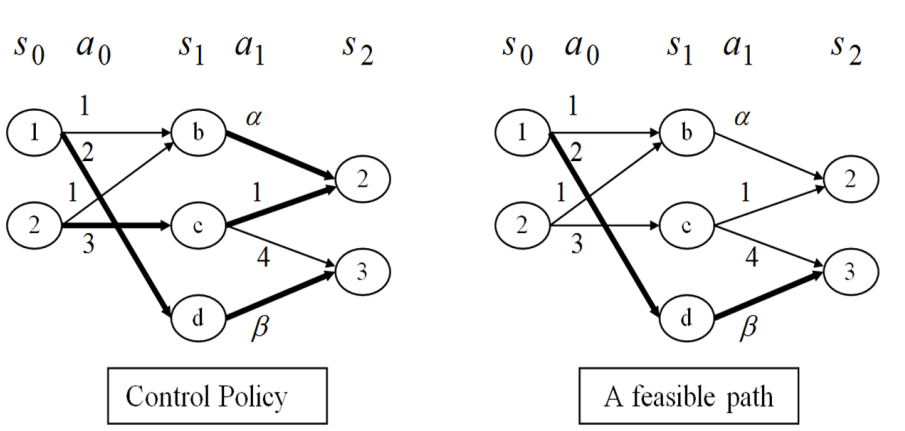
2.2.3 Control Policies

Definition 2.2. *A control policy, denoted π, specifies for each state a unique action to be taken at this state. Specifically, a control policy π is a sequence $\pi = (\pi_0, \ldots, \pi_{N-1})$ of decision functions, where $\pi_k : S_k \to A_k$, and $\pi_k(s) \in A_k(s)$. The action to be taken at time k in state s_k is given as $a_k = \pi_k(s_k)$*

Classes of control policies Observe that we allow the function π_k policy to depend on the time k. Such time-dependent policies are also called *non-stationary*. On the other hand, we confine ourselves here to policies that are:

- **Markovian**: The action a_k is a function of the current state s_k only, and not on previous states and actions. Non-Markovian policies: also called history-dependent policies, will be extensively used in the learning part of the course.
- **Deterministic**: More general policies may use randomization in the selection of actions. Such randomized policies are also used in learning algorithms, as well as in game problems.

Control policies and paths: As mentioned, a control policy specifies an action for each state, whereas a path specifies an action only for states along the path. This distinction is illustrated in the following figure.



Induced Path: A control policy π, together with an initial state s_0, specify a feasible path $h_N = (s_0, a_0, \ldots, s_{N-1}, a_{N-1}, s_N)$. This path may be computed recursively using $a_k = \pi_k(s_k)$ and $s_{k+1} = f_k(s_k, a_k)$, for $k = 0, 1, \ldots, N-1$.

Remark 2.4. *Suppose that for each state s_k, each action $a_k \in A_k(s_k)$ leads to a different state s_{k+1} (i.e., at most one edge connects any two states). We can then identify each action $a_k \in A_k(s)$ with the next state $s_{k+1} = f_k(s, a_k)$ it induces. In that case a path may be uniquely specified by the state sequence $(s_0, s_1, \ldots, s_N)$.*

2.2.4 Optimal Control Policies

Definition 2.3. *A control policy π is called **optimal** if, for each initial state s_0, it induces an optimal path h_N from s_0.*

An alternative definition can be given in terms of policies only. For that purpose, let $h_N(\pi; s_0)$ denote the path induced by the policy π from s_0. For a given return functional $R_N(h_N)$, denote $R_N(\pi; s_0) = R_N(h_N(\pi; s_0))$ That is, $R_N(\pi; s_0)$ is the cumulative return for the path induced by by π from s_0.

Definition 2.4. *A control policy π is called optimal if, for each initial state s_0, it holds that $R_N(\pi; s_0) \geq R_N(\tilde{\pi}; s_0)$ for any other policy $\tilde{\pi}$.*

Equivalence of the two definitions can be easily established (exercise). An optimal policy is often denoted by π^*.

The standard finite-horizon planning problem: Find a control policy π for the N-stage decision problem with the cumulative return (or cost) function.

The naive approach to finding an optimal policy: For finite models (i.e., finite state and action spaces), the number of feasible paths (or control policies) is finite. It is therefore possible, in principle, to enumerate all N-stage paths, compute the cumulative return for each one, and choose the one which gives the largest return. Let us evaluate the number of different paths and control policies. Suppose for simplicity that number of states at each stage is the same: $|X_k| = n$, and similarly the number of actions at each state is the same: $|A_k(x)| = m$ (with $m \leq n$) . The number of feasible N-stage paths for each initial state is seen to be m^N. The number of different policies is m^{nN}. For example, for a fairly small problem with $N = n = m = 10$, we obtain 10^{10} paths for each initial state (and 10^{11} overall), and 10^{100} control policies. Clearly it is not possible to enumerate them all.

Fortunately, Dynamic Programming offers a drastic simplification of the computational complexity for this problem.

2.3 Finite Horizon Dynamic Programming

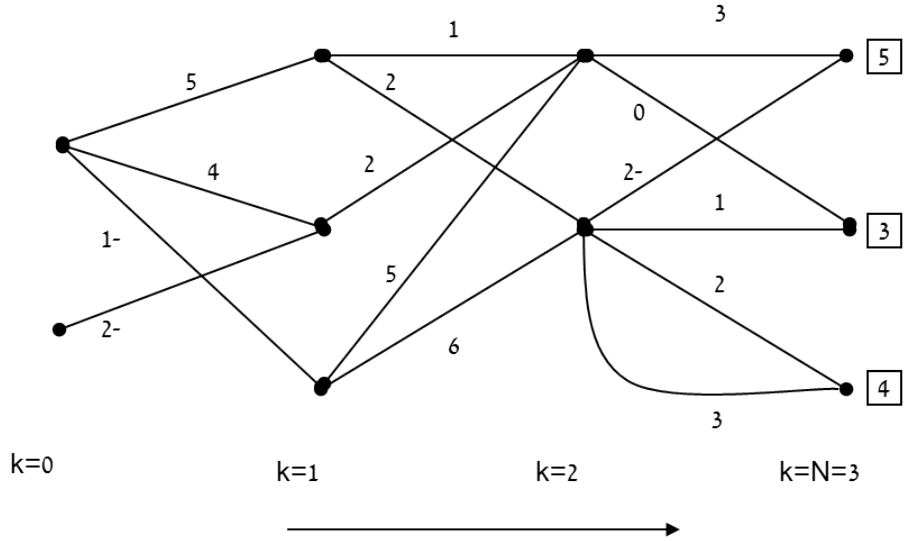
The Dynamic Programming (DP) algorithm breaks down the N-stage decision problem into N sequential single-stage optimization problems. This results in dramatic improvement in computation efficiency.

The DP technique for dynamic systems is based on a general observation called Bellman's Principle of Optimality. Essentially it states the following (for deterministic problems):

- **Any sub-path of an optimal path is itself an optimal path between its end point.**

Applying this principle recursively from the last stage backward, obtains the (backward) Dynamic Programming algorithm. Let us first illustrate the idea with following example.

Example 2.3. *Shortest path on a decision graph: Suppose we wish to find the shortest path (minimum cost path) from the initial node in N steps.*



The boxed values are the terminal costs at stage N, the other number are the link costs. Using backward recursion, we may obtain that the minimal path costs from the two initial states are 7 and 3, as well as the optimal paths and an optimal policy.

We can now describe the DP algorithm. Recall that we consider the dynamic system

$$s_{k+1} = f_k(s_k, a_k), \quad k = 0, 1, 2, \ldots, N-1$$

$$s_k \in S_k, \quad a_k \in A_k(s_k)$$

and we wish to maximize the cumulative return:

$$R_N = \sum_{k=0}^{N-1} r_k(s_k, a_k) + r_N(s_N)$$

The DP algorithm computes recursively a set of **value functions** $V_k : S_k \to \mathbb{R}$, where $V_k(s_k)$ is the value of an optimal sub-path $h_{k:N} = (s_k, a_k, \ldots, s_N)$ that starts at s_k.

Algorithm 2.1. ***Finite-horizon Dynamic Programming***

1. *Initialize the value function: $V_N(s) = r_N(s)$, $s \in S_N$.*
2. *Backward recursion: For $k = N-1, \ldots, 0$, compute*

$$V_k(s) = \max_{a \in A_k} \left\{ r_k(s, a) + V_{k+1}(f_k(s, a)) \right\}, \quad s \in S_k.$$

3. *Optimal policy: Choose any control policy $\pi = (\pi_k)$ that satisfies:*

$$\pi_k(s) \in \arg\max_{a \in A_k} \left\{ r_k(s,a) + V_{k+1}(f_k(s,a)) \right\}, \quad k = 0, \ldots, N-1.$$

Proposition 2.1. *The following holds for finite-horizon dynamic programming:*

1. *The control policy π computed above is an optimal control policy for the N-stage decision problem.*
2. *$V_0(s)$ is the optimal N-stage return from initial state $s_0 = s$:*

$$V_0(s) = \max_{\pi} R_N(\pi; s), \quad \forall s \in S_0.$$

We will provide a proof of this result in a later lecture, for the more general stochastic MDP model. For the time being, let us make the following observations:

1. The algorithm involves visits each state exactly once, proceeding backward in time. For each time instant (or stage) k, the value function $V_k(s)$ is computed for all states $s \in S_k$ before proceeding to stage $k-1$.
2. The recursive equation in part 2 of the algorithm, along with similar equations in the theory of DP, is called **Bellman's equation**.
3. Computational complexity: There is a total of nN states (excluding the final one), and in each we need m computations. Hence, the number of required calculations is mnN. For the example above with $m = n = N = 10$, we need $O(10^3)$ calculations.
4. A similar algorithm that proceeds forward in time (from $k = 0$ to $k = N$) can be devised. We note that this will not be possible for stochastic systems (i.e., the stochastic MDP model).
5. The celebrated **Viterbi algorithm** is an important instance of finite-horizon DP. The algorithm essentially finds the most likely sequence of states in a Markov chain (s_k) that is partially (or noisily) observed. The algorithm was introduced in 1967 for decoding convolution codes over noisy digital communication links. It has found extensive applications in communications, and is a basic computational tool in Hidden Markov Models (HMMs), a popular statistical model that is used extensively in speech recognition and bioinformatics, among other areas.

Historical Notes:

- Dynamic Programming was popularized in the 1950's and 1960's by **Richard Bellman** (1920-1984), an American mathematician (of Jewish descent). Bellman, who coined the term Dynamic Programming, formulated the general theory and proposed numerous applications in control and operations research.

- **Andrew Viterbi** (born 1935) is an American professor of electric engineer, a pioneer in the field of digital communications, and co-founder of Qualcomm Inc. (together with Irwin Jacobs). He has had close ties with the Technion, and has made many contributions to our department.

Chapter 3

Other Deterministic Dynamic Programming Algorithms

Dynamic Programming (DP) can be used to solve various computational problems that do not fall into the dynamic system framework which is our focus in this course. In this lecture we will briefly consider DP solutions to several such problems. Our treatment will be brief and informal.

This lecture is partly based on Chapter 15 of the textbook:

T. Cormen, C. Leliserson, R. Rivest and C. Stein, Introduction to Algorithms, 2nd ed., MIT Press, 2001.

Some of the problems below are nicely explained in `http://people.cs.clemson.edu/~bcdean/dp_practice/`

3.1 Specific Computational Problems

Dynamic Programming can be seen as a general approach for solving large computation problems by breaking them down into nested subproblems, and recursively combining the solutions to these subproblems. A key point is to organize the computation such that each subproblem is solved only once.

In many of these problems the recursive structure is not evident or unique, and its proper identification is part of the solution.

3.1.1 Maximum contiguous sum

Given: A (long) sequence of n real numbers $a_1, a_2, \ldots, a_n$ (both positive and negative).

Goal: Find $V^* = \max\limits_{1 \le i \le j \le n} \sum\limits_{\ell=i}^{j} a_\ell$.

An exhaustive search needs to examine $O(n^2)$ sums. Can this be done more efficiently?

DP Solution in linear time: Let $M_k = \max\limits_{1\le i\le k} \sum\limits_{\ell=i}^{k} a_\ell$ denote the maximal sum over all contiguous subsequences that end exactly at a_k.

Then

$$M_1 = a_1,$$

and

$$M_k = \max\{M_{k-1} + a_k,\ a_k\}.$$

We may compute M_k recursively for $k = 2 : n$. The required solution is given by

$$V^* = \max\{M_1, M_2, \ldots, M_n\},$$

This procedure requires only $O(n)$ calculations, i.e., linear time.

Note: The above computation gives the value of the maximal sum. If we need to know the range of elements that contribute to that sum, we need to keep track of the indices that maximized the various steps in the recursion.

3.1.2 Longest increasing subsequence

Given: A sequence of n real numbers $a_1, a_2, \ldots, a_n$.

Goal: Find the longest strictly increasing subsequence (not necessarily contiguous). E.g, for the sequence $(3, 1, 5, 3, 4)$, the solution is $(1, 3, 4)$. Observe that the number of subsequences is 2^n, therefore an exhaustive search is inefficient.

DP solution: Define L_j = longest strictly increasing subsequence ending at position j. Then

$$L_1 = 1,$$

$$L_j = \max\{L_i \ :\ i < j,\ a_i < a_j\} + 1, \quad j > 1$$

and the size of the longest subsequence is $L^* = \max_{1\le j\le n}(L_j)$.
Computing L_j recursively for $j = 1 : n$ gives the result with a running time[1] of $O(n^2)$.

3.1.3 An integer knapsack problem

Given: A knapsack (bag) of capacity $C > 0$, and a set of n items with respective sizes $S_1, \ldots, S_n$ and values (worth) $V_1, \ldots, V_n$. The sizes are positive and integer-valued.

[1] We note that this can be further improved to $O(n \log n)$. See Chapter 15 of the 'Introduction to Algorithms' textbook for more details.

Goal: Fill the knapsack to maximize the total value. That is, find the subset $A \subset \{1, \ldots, n\}$ of items that maximize

$$\sum_{i \in A} V_i,$$

subject to

$$\sum_{i \in A} S_i \leq C.$$

Note that the number of item subsets is 2^n.

DP solution: Let $M(i,k)$ = maximal value for filling exactly capacity k with items from the set $1:i$. If the capacity k cannot be matched by any such subset, set $M(i,k) = -\infty$. Also set $M(0,0) = 0$, and $M(0,k) = -\infty$ for $k \geq 1$. Then

$$M(i,k) = \max\{M(i-1,k)\,,\ M(i-1,k-S_i) + V_i\},$$

which can be computed recursively for $i = 1:n$, $k = 1:C$. The required value is obtained by $M^* = \max_{0 \leq k \leq C} M(n,k)$.
The running time of this algorithm is $O(nC)$. We note that the recursive computation of $M(n,k)$ requires $O(C)$ space. To obtain the indices of the terms in the optimal subset some additional book-keeping is needed, which requires $O(nC)$ space.

3.1.4 Longest Common Subsequence

Given: Two sequences (or strings) $X(1:m)$, $Y(1:n)$

Goal: A subsequence of X is the string that remains after deleting some number (zero or more) of elements of X. We wish to find the longest common subsequence (LCS) of X and Y, namely, a sequence of maximal length that is a subsequence of both X and Y.

For example:

$$X = \underline{A}V\underline{B}V\underline{A}M\underline{CD},$$
$$Y = \underline{A}Z\underline{B}Q\underline{AC}L\underline{D}.$$

DP solution: Let $c(i,j)$ denote the length of an LCS of the prefix subsequences $X(1:i)$, $Y(1:j)$. Set $c(i,j) = 0$ if $i = 0$ or $j = 0$. Then, for $i, j > 0$,

$$c(i,j) = \begin{cases} c(i-1,j-1)+1 & : \quad x(i)=y(j) \\ \max\{c(i,j-1), c(i-1,j)\} & : \quad x(i) \neq y(j) \end{cases}$$

We can now compute $c(i,j)$ recursively, using a row-first or column-first order. Computing $c(m,n)$ requires $O(mn)$ steps.

3.1.5 Further examples

The references mentioned above provide additional details on these problems, as well as a number of problems. These include, among others:

- The Edit-Distance problem: find the distance (or similarity) between two strings, by counting the minimal number of "basic operations" that are needed to transform one string to another. A common set of basic operations is: delete character, add character, change character. This problem is frequently encountered in natural language processing and bio-informatics (e.g., DNA sequencing) applications, among others.
- The Matrix-Chain Multiplication problem: Find the optimal order to compute a matrix multiplication $M_1 M_2 \cdots M_n$ (for non-square matrices).

3.2 Shortest Path on a Graph

The problem of finding the shortest path over a graph is one of the most basic ones in graph theory and computer science. We shall briefly consider here three major algorithms for this problem that are closely related to dynamic programming, namely: The Bellman-Ford algorithm, Dijkstra's algorithm, and A*.

3.2.1 Problem Statement

We introduce several definitions from graph-theory.

Definition 3.1. ***Weighted Graphs:*** *Consider a graph $G = (V, E)$ that consists of a finite set of vertices (or nodes) $V = \{v\}$ and a finite set of edges (or links) $E = \{e\}$. We will consider directed graphs, where each edge e is equivalent to an ordered pair $(v_1, v_2) \equiv (s(e), d(e))$ of vertices. To each edge we assign a real-valued weight (or cost) $w(e) = w(v_1, v_2)$.*

Definition 3.2. ***Path:*** *A path p on G from v_0 to v_k is a sequence $(v_0, v_1, v_2, \ldots, v_k)$ of vertices such that $(v_i, v_{i+1}) \in E$. A path is* ***simple*** *if all edges in the path are distinct. A* ***cycle*** *is a path with $v_0 = v_k$.*

Definition 3.3. ***Path length:*** *The length of a path $w(p)$ is the sum of the weights over its edges: $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.*

A **shortest path** from u to v is a path from u to v that has the smallest length $w(p)$ among such paths. Denote this minimal length as $d(u, v)$ (with $d(u, v) = \infty$ if no path exists from u to v). The shortest path problem has the following variants:

- Single pair problem: Find the shortest path from a given source vertex s to a given destination vertex t .

- Single source problem: Find the shortest path from a given source vertex s to all other vertices.
- Single destination: Find the shortest path to a given destination node t from all other vertices.
- All pair problem.

We note that the single-source and single-destination problems are symmetric and can be treated as one. The all-pair problem can of course be solved by multiple applications of the other algorithms, but there exist algorithms which are especially suited for this problem.

3.2.2 The Dynamic Programming Equation

The DP equation (or Bellman's equation) for the shortest path problem can be written as:

$$d(u,v) = \min\{w(u,u') + d(u',v) \ : \ (u,u') \in E\},$$

which holds for any pair of nodes u, v.
The interpretation: $w(u,u') + d(u',v)$ is the length of the path that takes one step from u to u', and then proceeds optimally. The shortest path is obtained by choosing the best first step. Another version, which singles out the last step, is $d(u,v) = \min\{d(u,v') + w(v',v) \ : (v',v) \in E\}$. We note that these equations are non-explicit, in the sense that the same quantities appear on both sides. These relations are however at the basis of the following explicit algorithms.

3.2.3 The Bellman-Ford Algorithm

This algorithm solves the single destination (or the equivalent single source) shortest path problem. It allows both positive and negative edge weights. Assume for the moment that there are *no negative-weight cycles* (why?).

Algorithm 3.1. ***Bellman-Ford Algorithm***

Input: *A weighted directed graph G, and destination node t.*

1. *Initialization:* $d[t] = 0$, $d[v] = \infty$ *for* $v \in V\backslash\{t\}$.
   % $d[v]$ holds the current shortest distance from v to t.
2. *for* $i = 1$ *to* $|V| - 1$,
   $\tilde{d}[v] = d[v], \quad v \in V\backslash\{t\}$ *% temporary buffer for d*
   for each vertex $v \in V\backslash\{t\}$,
   $q[v] = \min_u\{w(v,u) + \tilde{d}[u] : (v,u) \in E\}$
   $d[v] = \min\{q[v], d[v]\}$

3. for $v \in V \backslash \{t\}$,

if $d[v] < \infty$

set $\pi[v] \in \arg\min_u \{w(v,u) + \tilde{d}[u] : (v,u) \in E\}$

else

$\pi[v]$=*NULL*

4. return $\{d[v], \pi[v]\}$

The output of the algorithm is $d[v] = d(v,t)$, the weight of the shortest path from v to t, and the routing list π. A shortest path from vertex v is obtained from π by following he sequence: $v_1 = \pi[v]$, $v_2 = \pi[v_1]$, ...,$t = \pi[v_{k-1}]$. To understand the algorithm, we observe that after round i, $d[v]$ holds the length of the shortest path from v **in** i **steps or less**. (This can easily be verified by the results of the previous lecture.) Since the shortest path takes as most $|V| - 1$ steps, the above claim of optimality follows.

Remarks:

1. The running time of the algorithm is $O(|V| \cdot |E|)$. This is because in each round i of the algorithm, each edge e is involved in exactly one update of $d[v]$ for some v.

2. If $\{d[v]\}$ does not change at all at some round, then the algorithm may be stopped there.

3. In the version shown above, $\tilde{d}[v]$ is used to 'freeze' $d[v]$ for an entire round. The standard form of the algorithm actually uses $d[v]$ directly on the right-hand side.

4. We have assumed above that no negative-weight cycles exist. In fact the algorithm can be used to check for existence of such cycles: A negative-weight cycle exists if and only if $d[v]$ changes during an additional step ($i = |V|$) of the algorithm.

5. The basic scheme above can also be implemented in an asynchronous manner, where each node performs a local update of $d[v]$ at its own time. Further, the algorithm can be started from any initial conditions, although convergence can be slower. This makes the algorithm useful for distributed environments such as internet routing.

3.2.4 Dijkstra's Algorithm

Dijkstra's algorithm (introduced in 1959) provides a more efficient algorithm for the single-destination shortest path problem. This algorithm is restricted to non-negative link weights, i.e., $w(v,u) \geq 0$.

The algorithm essentially determines the minimal distance $d(v,t)$ of the vertices to the destination in order of that distance, namely the closest vertex first, then the second-closest, etc. The algorithm is roughly described below, with more details in the recitation.

The algorithm maintains a set S of vertices whose minimal distance to the destination has been determined. The other vertices are held in a queue Q. It proceeds as follows.

Algorithm 3.2. ***Dijkstra's Algorithm***

1. *Input: A weighted directed graph, and destination node t.*
2. *Initialization:*

   $d[t] = 0$
   $d[v] = \infty$ *for* $v \in V \backslash \{t\}$
   $\pi[v] = \phi$ *for* $v \in V$
   $S = \phi$
3. *while* $S \neq V$,
   choose $u \in V \backslash S$ *with minimal value* $d[u]$, *add it to* S
   for each vertex v *with* $(v, u) \in E$,
   if $d[v] > w(v, u) + d[u]$,
   set $d[v] = w(v, u) + d[u]$, $\pi[v] = u$
4. *return* $\{d[v], \pi[v]\}$

Remarks:

1. The Bellman-Form algorithm visits and updates each vertex of the graph up to $|V|$ times, leading to a running time of $O(|V| \cdot |E|)$. Dijkstra's algorithm visits each edge only once, which contributes $O(\,|E|)$ to the running time. The rest of the computation effort is spent on determining the order of node insertion to S.
2. The vertices in $V \backslash S$ need to be extracted in increasing order of $d[v]$. This is handled by a min-priority queue Q, and the complexity of the algorithm depends on the implementation of this queue.
3. With a naive implementation of the queue that simply keeps the vertices in some fixed order, each extract-min operation takes $O(|V|)$ time, leading to overall running time of $O(|V|^2 + |E|)$ for the algorithm. Using a basic (mean-heap) priority queue brings the running time to $O((|V| + |E|) \log |V|)$, and a more sophisticated one (Fibonacci heap) can bring it down to $O(|V| \log |V| + |E|)$.

3.2.5 Dijkstra's Algorithm for Single Pair Problems

For the single pair problem, Dijkstra's algorithm can be written in the Single Source Problem formulation, and terminated once the destination node is reached, i.e., when t is popped from the queue Q. From the discussion above, it is clear that the algorithm will terminate exactly when the shortest path between the source and destination is found.

Algorithm 3.3. ***Dijkstra's Algorithm (Single Pair Problem)***

1. *Input: A weighted directed graph, source node s, and destination node t.*
2. *Initialization:*

   $d[s] = 0$
   $d[v] = \infty$ *for* $v \in V \backslash \{s\}$
   $\pi[v] = \emptyset$ *for* $v \in V$
   $S = \emptyset$

3. *while* $S \neq V$,
   choose $u \in V \backslash S$ *with minimal value* $d[u]$, *add it to* S
   if $u == t$, *break*
   for each vertex v *with* $(u, v) \in E$,
   if $d[v] > d[u] + w(u, v)$,
   set $d[v] = d[u] + w(u, v)$, $\pi[v] = u$
4. *return* $\{d[v], \pi[v]\}$

3.2.6 From Dijkstra's Algorithm to A$*$

Dijkstra's algorithm expands vertices in the order of their distance from the source. When the destination is known (as in the single pair problem), it seems reasonable to bias the search order towards vertices that are closer to the goal.

The A* algorithm implements this idea through the use of a heuristic function $h[v]$, which is an estimate of the distance from vertex v to the goal. It then expands vertices in the order of $d[v] + h[v]$, i.e., the (estimated) length of the shortest path from s to t that passes through v.

Algorithm 3.4. ***A* Algorithm***

1. *Input: A weighted directed graph, source s, destination t, and heuristic function h.*
2. *Initialization:*

$d[s] = 0$

$d[v] = \infty$ *for* $v \in V \backslash \{s\}$

$\pi[v] = \emptyset$ *for* $v \in V$

$S = \emptyset$

3. *while* $S \neq V$,

   choose $u \in V \backslash S$ *with minimal value* $d[u] + h[u]$, *add it to* S

   if $u == t$, *break*

   for each vertex v *with* $(u, v) \in E$,

   if $d[v] > d[u] + w(u, v)$,

   set $d[v] = d[u] + w(u, v)$, $\pi[v] = u$

4. *return* $\{d[v], \pi[v]\}$

Obviously, we cannot expect the estimate $h(v)$ to be exact – if we knew the exact distance then our problem would be solved. However, it turns out that relaxed properties of h are required to guarantee the optimality of A*.

Definition 3.4. *A heuristic is said to be* ***consistent*** *if for every adjacent vertices* u, v *we have that*

$$w(v, u) + h[u] - h[v] \geq 0.$$

A heuristic is said to be ***admissible*** *if it is a lower bound of the shortest path to the goal, i.e., for every vertex* u *we have that*

$$h[u] \leq d^*[u, t],$$

where $d^*[u, v]$ *denotes the length of the shortest path between* u *and* v.

Remarks:

- It is easy to show that every consistent heuristic is also admissible (exercise: show it!). It is more difficult to find admissible heuristics that are not consistent. In path finding applications, a popular heuristic that is both admissible and consistent is the Euclidean distance to the goal.
- With a consistent heuristic, A* is guaranteed to find the shortest path in the graph. With an admissible heuristic, some extra bookkeeping is required to guarantee optimality.
- Actually, a stronger result can be shown for A*: for a given h, no other algorithm that is guaranteed to be optimal will explore less vertices during the search.

- The notion of admissibility is a type of *optimism*, and is required to guarantee that we don't settle on a suboptimal solution. Later in the course we will see that this idea plays a key role also in learning algorithms.
- We will show optimality for a consistent heuristic by showing that A* is equivalent to running Dijkstra's algorithm on a graph with modified weights.
    1. Define new weights $\hat{w}(u,v) = w(u,v) + h(v) - h(u)$. This transformation does not change the shortest path from s to t (show this!), and the new weights are non-negative due to the consistency property.
    2. The A* algorithm is equivalent to running Dijkstra's algorithm (for the single pair problem) with the weights $\hat{w}$, and defining $\hat{d}[v] = d[v]+h[v]$. The optimality of A* therefore follows from the optimality results for Dijsktra's algorithm.
- The idea of changing the cost function to make the problem easier to solve without changing the optimal solution is known as *cost shaping*, and also plays a role in learning algorithms.

3.3 Continuous Optimal Control

In this section we consider optimal control of continuous, deterministic, and fully observed systems in discrete time. In particular, consider the following problem:

$$\begin{aligned} &\min_{u_0,\dots,u_T} \sum_{t=0}^{T} c_t(x_t, u_t), \\ &s.t. \quad x_{t+1} = f_t(x_t, u_t), \end{aligned} \tag{3.1}$$

where the initial state x_0 is given. Here c_t is a (non-linear) cost function at time t, and f_t describes the (non-linear) dynamics at time t. We assume here that f_t and c_t are differentiable.

A simple approach for solving Problem 3.1 is using gradient based optimization. Note that we can expand the terms in the sum using the known dynamics function and initial state:

$$\begin{aligned} J(u_0, \dots, u_T) &= \sum_{t=0}^{T} c_t(x_t, u_t) \\ &= c_0(x_0, u_0) + c_1(f_0(x_0, u_0), u_1) + \cdots + c_T(f_{T-1}(f_{T-2}(\dots), u_{T-1}), u_T). \end{aligned}$$

Using our differentiability assumption, we know $\frac{\partial f_t}{\partial x_t}, \frac{\partial f_t}{\partial u_t}, \frac{\partial c_t}{\partial x_t}, \frac{\partial c_t}{\partial u_t}$. Thus, using repeated application of the chain rule, we can calculate $\frac{\partial J}{\partial u_t}$, and optimize J using gradient descent. There are, however, two potential issues with this approach. The first is that we will only

be guaranteed a locally optimal solution. The second is that in practice, for large T, the repeated calculation in the gradient often suffers from numerical instability.

We will now show a different approach. We will first show that for linear systems and quadratic costs, Problem 3.1 can be solved using dynamic programming. This problem is often called a Linear Quadratic Regulator (LQR). We will then show how to extend the LQR solution to non-linear problems using linearization, resulting in an iterative LQR algorithm (iLQR).

3.3.1 Linear Quadratic Regulator

We now restrict our attention to linear-quadratic problems of the form:

$$\begin{aligned} &\min_{u_0,\ldots,u_T} \sum_{t=0}^{T} c_t(x_t, u_t), \\ &s.t. \quad x_{t+1} = A_t x_t + B_t u_t, \\ &\qquad c_t = x_t^\top Q_t x_t + u_t^\top R_t u_t, \forall t = 0, \ldots, T-1, \\ &\qquad c_T = x_t^\top Q_T x_t. \end{aligned} \tag{3.2}$$

where x_0 is given, and $Q_t = Q_t^\top \geq 0$, $R_t = R_t^\top > 0$ are state-cost and control-cost matrices.

We will solve Problem 3.2 using dynamic programming. Let $V_t(x)$ denote the value function of a state at time t, that is, $V_t(x) = \min_{u_t,\ldots,u_T} \sum_{t'=t}^{T} c_{t'}(x_{t'}, u_{t'}) \quad \text{s.t.} \quad x_t = x$.

Proposition 3.1. *The value function has a quadratic form: $V_t(x) = x^\top P_t x$, and $P_t = P_t^\top$.*

Proof. We prove by induction. For $t = T$, this holds by definition, as $V_T(x) = x^\top Q_T x$. Now, assume that $V_{t+1}(x) = x^\top P_{t+1} x$. We have that

$$\begin{aligned} V_t(x) &= \min_{u_t} x^\top Q_t x + u_t^\top R_t u_t + V_{t+1}(A_t x + B_t u_t) \\ &= \min_{u_t} x^\top Q_t x + u_t^\top R_t u_t + (A_t x + B_t u_t)^\top P_{t+1}(A_t x + B_t u_t) \\ &= x^\top Q_t x + (A_t x)^\top P_{t+1}(A_t x) + \min_{u_t} u_t^\top (R_t + B_t^\top P_{t+1} B_t) u_t + 2(A_t x)^\top P_{t+1}(B_t u_t) \end{aligned}$$

The objective is quadratic in u_t, and solving the minimization gives

$$u_t^* = -(R_t + B_t^\top P_{t+1} B_t)^{-1} B_t^\top P_{t+1} A_t x.$$

Substituting back u_t^* in the expression for $V_t(x)$ gives a quadratic expression in x. □

From the construction in the proof of Proposition 3.1 one can recover the sequence of optimal controllers u_t^*. By substituting the optimal controls in the forward dynamics equation, one can also recover the optimal state trajectory.

Remarks:

1. The DP solution is *globally* optimal for the LQR problem. Interestingly, the computational complexity is polynomial in the *dimension* of the state, and linear in the time horizon. This is in contrast to the curse of dimensionality, and is due to the special structure of the dynamics and cost.

2. Note that the DP computation resulted in a sequence of *linear feedback controllers*. It turns out that these controllers are also optimal in the presence of Gaussian noise added to the dynamics.

3. A similar derivation holds for the system:

$$\begin{aligned} &\min_{u_0,\ldots,u_T} \sum_{t=0}^{T} c_t(x_t, u_t), \\ &s.t. \quad x_{t+1} = A_t x_t + B_t u_t + C_t, \\ &\qquad c_t = [x_t, u_t]^\top W_t [x_t, u_t] + Z_t [x_t, u_t] + Y_t, \forall t = 0, \ldots, T. \end{aligned}$$

   In this case, the optimal control is of the form $u_t^* = K_t x + k_t$, for some K_t and k_t.

3.3.2 Iterative LQR

We now return to the original non-linear problem 3.1. If we linearize the dynamics and quadratize the cost – we can plug in the LQR solution we obtained above. Namely, given some reference trajectory $\hat{x_0}, \hat{u_0}, \ldots, \hat{x_T}, \hat{u_T}$, we apply a Taylor approximation:

$$\begin{aligned} f_t(x_t, u_t) &\approx f_t(\hat{x}_t, \hat{u}_t) + \nabla_{x_t,u_t} f_t(\hat{x}_t, \hat{u}_t)[x_t - \hat{x}_t, u_t - \hat{u}_t] \\ c_t(x_t, u_t) &\approx c_t(\hat{x}_t, \hat{u}_t) + \nabla_{x_t,u_t} c_t(\hat{x}_t, \hat{u}_t)[x_t - \hat{x}_t, u_t - \hat{u}_t] \\ &\quad + \frac{1}{2}[x_t - \hat{x}_t, u_t - \hat{u}_t]^\top \nabla^2_{x_t,u_t} c_t(\hat{x}_t, \hat{u}_t)[x_t - \hat{x}_t, u_t - \hat{u}_t]. \end{aligned} \tag{3.3}$$

If we define $\delta_x = x - \hat{x}$, $\delta_u = u - \hat{u}$, then the Taylor approximation gives an LQR problem for δ_x, δ_u. It's optimal controller is $u_t^* = K_t(x_t - \hat{x}_t) + k_t + \hat{u}_t$. By running this controller on the non-linear system, we obtain a new reference trajectory. Also note that the controller $u_t^* = K_t(x_t - \hat{x}_t) + \alpha k_t + \hat{u}_t$ for $\alpha \in [0, 1]$ smoothly transitions from the previous trajectory ($\alpha = 0$) to the new trajectory ($\alpha = 1$) (show that!). Therefore we can interpret α as a step size, to guarantee that we stay within the Taylor approximation limits.

The iterative LQR algorithm works by applying this approximation iteratively:

1. Initialize a control sequence $\hat{u_0}, \ldots, \hat{u_T}$ (e.g., by zeros).

2. Run a forward simulation of the controls in the nonlinear system to obtain a state trajectory $\hat{x_0}, \ldots, \hat{x_T}$.

3. Linearize the dynamics and quadratize the cost (Eq. 3.3), and solve using LQR.

4. By running a forward simulation of the control $u_t^* = K_t(x_t - \hat{x}_t) + \alpha k_t + \hat{u}_t$ on the non-linear system, perform a line search for the optimal α according to the non-linear cost.

5. For the found α, run a forward simulation to obtain a new trajectory $\hat{x_0}, \hat{u_0}, \ldots, \hat{x_T}, \hat{u_T}$. Go to step 3.

In practice, the iLQR algorithm is typically much more stable and efficient than the naive gradient descent approach.

3.4 Exercises

Exercise 3.1 (**Counting**). Given an integer number X, we would like to count in how many ways it can be represented as a sum of N natural numbers (with relevance to order) marked by $\Psi_N(X)$. For example, $\Psi_2(3) = 2$ since: $3 = 1 + 2 = 2 + 1$.

1. Find the following: $\Psi_1(2), \Psi_2(4), \Psi_3(2), \Psi_N(N)$ and $\Psi_1(X)$.

2. Find a recursive equation for $\Psi_N(X)$.

3. Write a code in Matlab for finding $\Psi_N(X)$ using dynamic programming.

   (a) What is the time and memory complexity of your algorithm?

   (b) Find $\Psi_{12}(800)$.

4. Now assume each natural number i is associated with some cost c_i. For a given X, N we are interested in finding the lowest cost combination of natural numbers $\{x_i\}_{i=1}^N$ satisfying $\sum_{i=1}^N x_i = X$.

   (a) Formulate the problem as a finite horizon decision problem: Define the state space, the action space and the cumulative cost function.

   (b) Bound the complexity of finding the best combination.

Exercise 3.2 (**Language model**). In the city of Bokoboko the locals use a language with only 3 letters ('B','K','O'). After careful inspection of this language, researchers have reached two conclusions:

I. Every word starts with the letter 'B'.

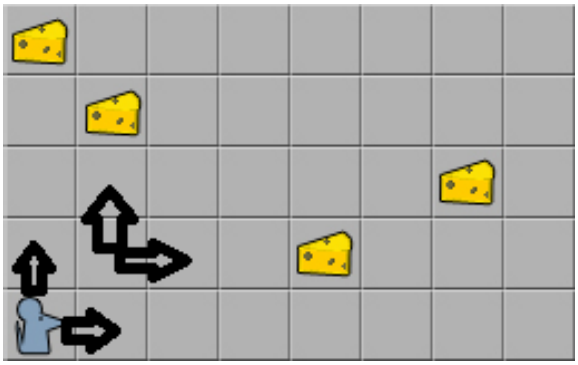
II. Every consecutive letter is distributed only according to the previous letter as follows:

$$P\left(l_{t+1}|l_t\right) = \begin{matrix} B \\ K \\ O \\ - \end{matrix} \begin{bmatrix} 0.1 & 0.325 & 0.25 & 0.325 \\ 0.4 & 0 & 0.4 & 0.2 \\ 0.2 & 0.2 & 0.2 & 0.4 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Where '-' represents the end of a word. For example, the probability of the word 'bko' is given by $0.325 \cdot 0.4 \cdot 0.4 = 0.052$.

1. Find the probability of the following words: 'Bob', 'Koko', 'B', 'Bokk','Boooo'.
2. We wish to find the most probable word in the language of length K.
   (a) Formulate the problem as a finite horizon decision problem: Define the state space, the action space and the multiplicative cost function.
   (b) Bound the complexity of finding the best combination.
   (c) Find a reduction from the given problem to an analogous problem with additive cost function instead.
   (d) Explain when each approach (multiplicative vs. additive) is preferable.
   Hint: Think of the consequence of applying the reduction on the memory representation of a number in a standard operating system.
   (e) Write a code in Matlab which finds the most probable word of a given size using dynamic programming. What is the most probable word of size 5?

Exercise 3.3 (Path Planning)**.** Moses the mouse starts his journey at the south west room in a $M \times N$ rectangular apartment with $M \cdot N$ rooms of size 1×1, some of which contain cheese. After his rare head injury in the mid-scroll war, Moses can only travel north or east. An illustration of Moses's life for $M = 5, N = 8$ is given in the following figure.



Being a mouse and all, Moses wants to gather as much cheese as possible until he reaches the north-east room of the apartment.

1. Formulate the problem as a finite horizon decision problem: Define the state space, the action space and the cumulative cost function.

2. What is the horizon of the problem?

3. How many possible trajectories are there? How does the number of trajectories behaves as a function of N when $M = 2$? How does it behave as a function of N when $M = N$?

4. Aharon, Moses's long lost war-buddy woke up confused next to Moses and decided to join him in his quest (needless to say, both mice suffer the same rare head injury).

   (a) Explain what will happen if both mice ignore each other's existence and act 'optimal' with respect to the original problem.

   (b) Assume both mice decided to coordinate their efforts and split the loot. How many states and actions are there now?

   (c) Now their entire rarely-head-injured division has joined the journey. Assume there's a total of K mice, how many states and actions are there now?

Exercise 3.4 (**MinMax dynamic programming**)**.** In this problem we consider an adversarial version of finite horizon dynamic programming, which is suitable for solving 2-player games.

In this setting, at time $k \in \{0, 1, 2, \ldots, N-1\}$ the system is at state $s_k \in S_k$, the agent chooses an action $a_k \in A_k(s_k)$ according to the agent policy $\pi_k^a(s_k)$, and subsequently the opponent chooses an action b_k from the set of allowable opponent actions $B_k(s_k, a_k)$, according to the opponent policy $\pi_k^b(s_k, a_k)$.

The system then transitions to a new state according to:

$$s_{k+1} = f_k(s_k, a_k, b_k), \quad k = 0, 1, 2, \ldots, N-1.$$

The instantaneous reward is denoted by $r(s_k, a_k, b_k)$, and, for an N-stage path $h_N = (s_0, a_0, b_0 \ldots, s_{N-1}, a_{N-1}, b_{N-1}, s_N)$ the cumulative reward is

$$R_N(h_N) = \sum_{k=0}^{N-1} r_k(s_k, a_k, b_k) + r_N(s_N).$$

Given s_0, the agent's goal is to find a policy π_a^* that maximizes the worst-case cumulative reward:

$$\pi_a^* \in \arg\max_{\pi_a} \left\{ \min_{\pi_b} \{R_N(h_N)\} \right\}.$$

1. Formulate a dynamic programming algorithm for this problem. Explain what the value function represents in this case.

	Eilat	Ashdod	Beer Sheva	Haifa	Jerusalem	Nazereth	Netanya	Petah Tikva	Rehovot	Tel Aviv
Eilat	0	328	246	-	323	-	381	-	327	-
Ashdod	328	0	82	-	-	-	-	47	24	42
Beer Sheva	246	82	0	-	89	-	-	106	82	103
Haifa	-	-	-	0	145	40	64	91		95
Jerusalem	323	-	89	145	0	142	-	60	59	62
Nazereth	-	-	-	40	142	0	75	-	-	-
Netanya	381	-	-	64	-	75	0	30	54	33
Petah Tikva	-	47	106	91	60	-	30	0	29	10
Rehovot	327	24	82	-	59	-	54	29	0	21
Tel Aviv	-	42	103	95	62	-	33	10	21	0

Table 3.1: City-distances

2. What is the computational complexity of your algorithm?

3. Could this approach be used to solve the game of tic-tac-toe? Explain what are the states, actions and rewards for this game.

4. Could this approach be used to solve the game of chess? Explain.

Exercise 3.5 (**SSSP**)**.** This exercise concerns the SSSP problem.

1. Give an example of a graph that does not contain negative cycles, but for which Dijkstra's algorithm fails. Prove that your graph indeed does not contain any negative cycles.

2. Consider the following road distances table (Excel file available on the course webpage). Each number in the table represents a road between the two cities of the mentioned distance.

   (a) How many nodes and edges are there in the induced graph?

   (b) What is the shortest path between Tel Aviv and Haifa? What about Jerusalem and Nazereth?

   (c) Program Dijkstra's algorithm and find the shortest path from each city to each other city. What is the time complexity of your solution in terms of the number of nodes and edges in the graph?

   (d) Find the shortest route that starts at Jerusalem, visits all other cities and then returns to Jerusalem. What is the time complexity of your solution in terms of the number of nodes and edges in the graph?

Chapter 4

Markov Decision Processes

In the previous lectures we considered multi-stage decision problems for *deterministic* systems. In many problems of interest, the system dynamics also involves *randomness*, which leads us to stochastic decision problems. In this lecture we introduce the basic model of Markov Decision Processes, which will be considered is the rest of the course.

4.1 Markov Chains: A Reminder

A Markov chain $\{X_t,\ t = 0, 1, 2, \ldots\}$, with $X_t \in X$, is a discrete-time stochastic process, over a finite or countable state-space X, that satisfies the following Markov property:

$$\mathcal{P}(X_{t+1} = j | X_t = i, X_{t-1}, \ldots X_0) = \mathcal{P}(X_{t+1} = j | X_t = i).$$

We focus on time-homogeneous Markov chains, where

$$\mathcal{P}(X_{t+1} = j | X_t = i) = \mathcal{P}(X_1 = j | X_0 = i) \stackrel{\Delta}{=} p_{ij}.$$

The p_{ij}'s are the transition probabilities, which satisfy $p_{ij} \geq 0, \quad \sum_{i \in X} p_{ij} = 1 \ \ \forall j$. The matrix $P = (p_{ij})$ is the transition matrix.

Given the initial distribution p_0 of X_0, namely $p(X_0 = i) = p_0(i)$, we obtain the finite-dimensional distributions:

$$\mathcal{P}(X_0 = i_0, \ldots, X_t = i_t) = p_0(i) p_{i_0 i_1} \cdot \ldots \cdot p_{i_{t-1} i_t}.$$

Define $p_{ij}^{(m)} = \mathcal{P}(X_m = j | X_0 = i)$, the m-step transition probabilities. It is easy to verify that $p_{ij}^{(m)} = [P^m]_{ij}$ (here P^m is the m-th power of the matrix P).

State classification:

- State j is accessible from i ($i \to j$) if $p_{ij}^{(m)} > 0$ for some $m \geq 1$.

- i and j are communicating states (or communicate) if $i \to j$ and $j \to i$.
- A communicating class (or just class) is a maximal collection of states that communicate.
- The Markov chain is irreducible if all states belong to a single class (i.e., all states communicate with each other).
- State i is periodic with period $d \geq 2$ if $p_{ii}^{(m)} = 0$ for $m \neq \ell d$ and $p_{ii}^{(m)} > 0$ for $m = \ell d$, $\ell = 1, 2, \ldots$. Periodicity is a class property: all states in the same class have the same period.

Recurrence:

- State i is recurrent if $\mathcal{P}(X_t = i \text{ for some } t \geq 1 | X_0 = i) = 1$. Otherwise, i is transient.
- State i is recurrent if and only if $\sum_{m=1}^{\infty} p_{ii}^{(m)} = \infty$.
- Recurrence is a class property.
- If i and j are in the same recurrent class, then j is (eventually) reached from i with probability 1: $\mathcal{P}(X_t = j \text{ for some t} \geq 1 | X_0 = i) = 1$.
- Let T_i be the return time to state i (number of stages required for (X_t) to return to i). If i is a recurrent state, then $T_i < \infty$ w.p. 1.
- State i is positive recurrent if $E(T_i) < \infty$, and null recurrent if $E(T_i) = \infty$. If the state space is finite, all recurrent states are positive recurrent.

Invariant Distribution: The probability vector $\pi = (\pi_i)$ is an invariant distribution or stationary distribution for the Markov chain if $\pi P = \pi$, namely

$$\pi_j = \sum\nolimits_i \pi_i p_{ij} \quad \forall j.$$

Clearly, if $X_t \sim \pi$ then $X_{t+1} \sim \pi$. If $X_0 \sim \pi$, then the Markov chain (X_t) is a stationary stochastic process.

Theorem 4.1 (Recurrence of finite Markov chains). *Let (X_t) be an irreducible, aperiodic Markov chain over a finite state space X. Then the following properties hold:*

1. *All states are **positive recurrent***
2. *There exists a **unique stationary distribution** π^**
3. ***Convergence** to the stationary distribution:* $\lim_{t\to\infty} p_{ij}^{(t)} = \pi_j \quad (\forall j)$

4. ***Ergodicity***: *For any finite* f: $\lim_{t\to\infty}\frac{1}{t}\sum_{s=0}^{t-1} f(X_s) = \sum_i \pi(i)f(i) \stackrel{\Delta}{=} \pi \cdot f$.

For countable Markov chains, there are other possibilities.

Theorem 4.2 (**Countable Markov chains**). *Let* (X_t) *be an irreducible and a-periodic Markov chain over a countable state space* X. *Then:*

1. *Either (i) all states are positive recurrent, or (ii) all states are null recurrent, or (iii) all states are transient.*
2. *If (i) holds, then properties (2)-(4) of the previous Theorem hold as well.*
3. *Conversely, if there exists a stationary distribution* π *then properties (1)-(4) are satisfied.*

Reversible Markov chains: Suppose there exists a probability vector $\pi = (\pi_i)$ so that

$$\pi_i p_{ij} = \pi_j p_{ji}, \qquad i, j \in X. \tag{4.1}$$

It is then easy to verify by direct summation that π is an invariant distribution for the Markov chain defined by (p_{ij}). The equations (4.1) are called the *detailed balance equations*. A Markov chain that satisfies these equations is called reversible.

Example 4.1 (**Discrete-time queue**). *Consider a discrete-time queue, with queue length* $X_t \in \mathbb{N}_0 = \{0, 1, 2, \dots\}$. *At time instant* t, A_t *new jobs arrive, and then up to* S_t *jobs can be served, so that*

$$X_{t+1} = (X_t + A_t - S_t)^+.$$

Suppose that (S_t) *is a sequence of i.i.d. RVs, and similarly* (A_t) *is a sequence of i.i.d. RVs, with* (S_t), (A_t) *and* X_0 *mutually independent. It may then be seen that* $(X_t,\ t \geq 0)$ *is a Markov chain. Suppose further that each* S_t *is a Bernoulli RV with parameter* q, *namely* $P(S_t = 1) = q$, $P(S_t = 0) = 1 - q$. *Similarly, let* A_t *be a Bernoulli RV with parameter* p. *Then*

$$p_{ij} = \begin{cases} p(1-q) & : \quad j = i+1 \\ (1-p)(1-q) + pq & : \quad j = i, \ \ i > 0 \\ (1-p)q & : \quad j = i-1, \ \ i > 0 \\ (1-p) + pq & : \quad j = i = 0 \\ 0 & : \quad \text{otherwise} \end{cases}$$

Denote $\lambda = p(1-q)$, $\mu = (1-p)q$, *and* $\rho = \lambda/\mu$. *The detailed balance equations for this case are:*

$$\pi_i \lambda = \pi_{i+1}\mu, \qquad i \geq 0$$

These equations have a solution with $\sum_i \pi_i = 1$ *if and only if* $\rho < 1$. *The solution is* $\pi_i = \pi_0 \rho^i$, *with* $\pi_0 = 1 - \rho$. *This is therefore the stationary distribution of this queue.*

4.2 Controlled Markov Chains

A Markov Decision Process consists of two main parts:

1. A controlled dynamic system, with stochastic evolution.
2. A performance objective to be optimized.

In this section we describe the first part, which is modeled as a controlled Markov chain.

Consider a controlled dynamic system, defined over:

- A discrete time axis $\mathbf{T} = \{0, 1, \ldots, T-1\}$ (finite horizon), or $\mathbf{T} = \{0, 1, 2, \ldots\}$ (infinite horizon). To simplify the discussion we refer below to the infinite horizon case, which can always be "truncated" at T if needed.
- A finite state space S, where $S_t \subset S$ is the set of possible states at time t .
- A finite action set A, where $A_t(s) \subset A$ is the set of possible actions at time t and state $s \in S_t$.

State transition probabilities:

- Suppose that at time t we are in state $s_t = s$, and choose an action $a_t = a$. The next state $s_{t+1} = s'$ is then determined randomly according to a probability distribution $p_t(\cdot|s, a)$ on S_{t+1}. That is,

$$\mathcal{P}(s_{t+1} = s'|s_t = s, a_t = a) = p_t(s'|s, a), \qquad s' \in S_{t+1}$$

- The probability $p_t(s'|s, a)$ is the *transition probability* from state s to state s' for a given action a. We naturally require that $p_t(s'|s, a) \geq 0$, and $\sum_{s' \in S_{t+1}} p_t(s'|s, a) = 1$ for all $s \in S_t, a \in A_t(s)$.
- Implicit in this definition is the controlled-Markov property:

$$\mathcal{P}(s_{t+1} = s'|s_t, a_t) = \mathcal{P}(s_{t+1} = s'|s_t, a_t, \ldots, s_0, a_0)$$

- The set of probability distributions

$$P = \{p_t(\cdot|s, a) \quad : \quad s \in S_t, a \in A_t(s), t \in \mathbf{T}\}$$

  is called the *transition law* or *transition kernel* of the controlled Markov process.

Stationary Models: The controlled Markov chain is called stationary or time-invariant if the transition probabilities do not depend on the time t. That is:

$$\forall t, \quad p_t(s'|s, a) \equiv p(s'|s, a), \;\; S_t \equiv S, \;\; A_t(s) \equiv A(s).$$

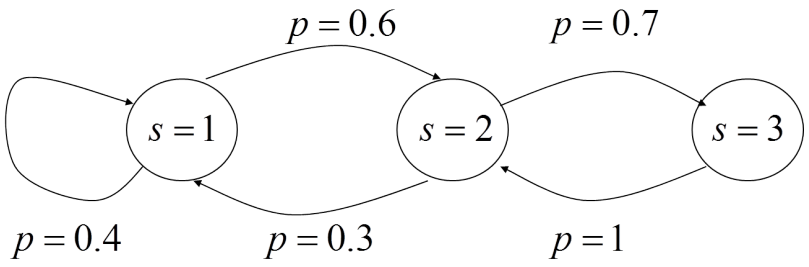


Figure 4.1: Markov chain

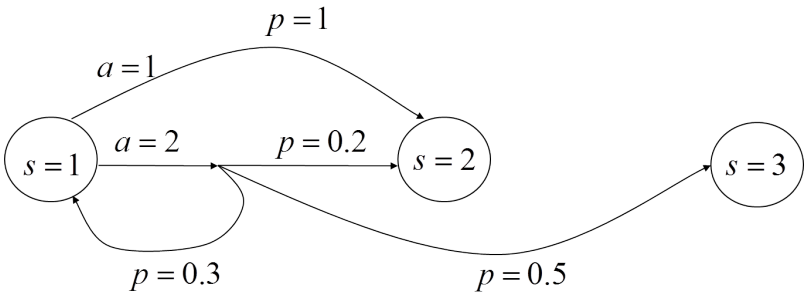


Figure 4.2: Controlled Markov chain

Graphical Notation: The state transition probabilities of a Markov chain are often illustrated via a state transition diagram, such as in Figure 4.1.

A graphical description of a controlled Markov chain is a bit more complicated because of the additional action variable. We obtain the diagram (drawn for state $s = 1$ only, and for a given time t) in Figure 4.2, reflecting the following transition probabilities:

$$p(s' = 2|s = 1, a = 1) = 1$$
$$p(s'|s = 1, a = 2) = \begin{cases} 0.3 & : \quad s' = 1 \\ 0.2 & : \quad s' = 2 \\ 0.5 & : \quad s' = 3 \end{cases}$$

State-equation notation: The stochastic state dynamics can be equivalently defined in terms of a state equation of the form

$$s_{t+1} = f_t(s_t, a_t, w_t),$$

where w_t is a random variable. If $(w_t)_{t\geq 0}$ is a sequence of independent RVs, and further each w_t is independent of the "past" $(s_{t-1}, a_{t-1}, \ldots s_0)$, then $(s_t, a_t)_{t\geq 0}$ is a controlled Markov

process. For example, the state transition law of the last example can be written in this way, using $w_t \in \{4, 5, 6\}$, with $p_w(4) = 0.3,\ p_w(5) = 0.2,\ p_w(6) = 0.5$ and, for $s_t = 1$:

$$\begin{array}{l} f_t(1, 1, w_t) = 2 \\ f_t(1, 2, w_t) = w_t - 3 \end{array} .$$

The state equation notation is especially useful for problems with continuous state space, but also for some models with discrete states.

Control Policies

- A general or **history-dependent** control policy $\pi = (\pi_t)_{t \in \mathbf{T}}$ is a mapping from each possible history $h_t = (s_0, a_0, \ldots, s_{t-1}, a_{t-1}, s_t)$, $t \in \mathbf{T}$, to an action $a_t = \pi_t(h_t) \in A_t$. We denote the set of general policies by Π_G.
- A **Markov** control policy π is allowed to depend on the current state and time only: $a_t = \pi_t(s_t)$. We denote the set of Markov policies by Π_M.
- For stationary models, we may define **stationary** control policies that depend on the current state alone. A stationary policy is defined by a single mapping $\pi : S \to A$, so that $a_t = \pi(s_t)$ for all $t \in \mathbf{T}$. We denote the set of stationary policies by Π_S.
- Evidently, $\Pi_G \supset \Pi_M \supset \Pi_S$.

Randomized Control policies

- The control policies defined above specify deterministically the action to be taken at each stage. In some cases we want to allow for a random choice of action.
- A general randomized control policy assigns to each possible history h_t a probability distribution $\pi_t(\cdot|h_t)$ over the action set A_t. That is, $prob\{a_t = a|h_t\} = \pi_t(a|h_t)$. We denote the set of general randomized policies by Π_{GR}.
- Similarly, we can define the set Π_{MR} of Markov randomized control policies, where $\pi_t(\cdot|h_t)$ is replaced by $\pi_t(\cdot|s_t)$, and the set Π_{SR} of stationary randomized control policies, where $\pi_t(\cdot|s_t)$ is replaced by $\pi(\cdot|s_t)$.
- Note that the set Π_{GR} includes all other policy sets as special cases.

The Induced Stochastic Process Let $p_0 = \{p_0(s), s \in S_0\}$ be a probability distribution for the initial state s_0. A control policy $\pi \in \Pi_{GR}$, together with the transition law

$P = \{p_t(s'|s,a)\}$ and the initial state distribution $p_0 = (p_0(s),\ s \in S_0)$, induce a probability distribution over any finite state-action sequence $h_T = (s_0, a_0, \ldots, s_{T-1}, a_{T-1}, s_T)$, given by

$$P(h_T) = p_0(s_0) \prod_{t=0}^{T-1} p(s_{t+1}|s_t, a_t)\pi_t(a_t|h_t).$$

To see this, observe the recursive relation:

$$\begin{aligned} P(h_{t+1}) = P(h_t, a_t, s_{t+1}) &= P(s_{t+1}|h_t, a_t)P(a_t|h_t)P(h_t) \\ &= p_t(s_{t+1}|s_t, a_t)\pi_t(a_t|h_t)P(h_t). \end{aligned}$$

In the last step we used the conditional Markov property of the controlled chain: $P(s_{t+1}|h_t, a_t) = p_t(s_{t+1}|s_t, a_t)$, and the definition of the control policy π. The required formula follows by recursion.

Therefore, the state-action sequence $h_\infty = (s_k, a_k)_{k\geq 0}$ can now be considered a stochastic process. We denote the probability law of this stochastic process by $P^{\pi,p_0}(\cdot)$. The corresponding expectation operator is denoted by $E^{\pi,p_0}(\cdot)$. When the initial state s_0 is deterministic (i.e., $p_0(s)$ is concentrated on a single state s), we may simply write $P^{\pi,s}(\cdot)$ or $P^\pi(\cdot|s_0 = s)$.

Under a Markov control policy, the state sequence $(s_t)_{t\geq 0}$ becomes a *Markov chain*, with transition probabilities

$$P(s_{t+1} = s'|s_t = s) = \sum\nolimits_{a\in A_t} p_t(s'|s,a)\pi_t(a|s).$$

Exercise 4.1. Prove this!

If the controlled Markov chain is stationary (time-invariant) and the control policy is stationary, then the induced Markov chain is stationary as well.

Remark 4.1. *For most non-learning optimization problems, Markov policies suffice to achieve the optimum.*

Remark 4.2. *Implicit in these definitions of control policies is the assumption that the current state s_t can be fully observed before the action a_t is chosen . If this is not the case we need to consider the problem of a Partially Observed MDP (POMDP), which is more involved.*

4.3 Performance Criteria

4.3.1 Finite Horizon Problems

Consider the finite-horizon problem, with a fixed time horizon T. As in the deterministic case, we are given a running reward function $r_t = \{r_t(s,a) : s \in S_t, a \in A_t\}$ for $0 \leq t \leq$

$T-1$, and a terminal reward function $r_T = \{r_T(s) : s \in S_T\}$. The obtained rewards are then $R_t = r_t(s_t, a_t)$ at times $t \leq T-1$, and $R_T = r_T(s_T)$ at the last stage. Our general goal is to maximize the cumulative return:

$$\sum_{t=0}^{T} R_t = \sum_{t=0}^{T-1} r_t(s_t, a_t) + r_T(s_T).$$

However, since the system is stochastic, the cumulative return will generally be random, and we need to specify in which sense to maximize it. A natural first option is to consider the expected value of the return. That is, define:

$$J_T^{\pi}(s) = E^{\pi}(\sum_{t=0}^{T} R_t | s_0 = s) \equiv E^{\pi,s}(\sum_{t=0}^{T} R_t)$$

Here π is the control policy as defined above, and s denotes the initial state. Hence, $J_T^{\pi}(s)$ is the expected cumulative return under the control policy π. Our goal is to find an optimal control policy that maximized $J_T^{\pi}(s)$.

Remarks:

1. Dependence on the next state: In some problems, the obtained reward may depend on the next state as well: $R_t = \tilde{r}_t(s_t, a_t, s_{t+1})$. For control purposes, when we only consider the expected value of the reward, we can reduce this reward function to the usual one by defining

$$r_t(s,a) \stackrel{\Delta}{=} E(R_t | s_t = s, a_t = a) \equiv \sum\nolimits_{s' \in S} p(s'|s,a)\tilde{r}_t(s,a,s')$$

2. Random rewards: The reward R_t may also be random, namely a random variable whose distribution depends on (s_t, a_t). This can also be reduced to our standard model for planning purposes by looking at the expected value of R_t, namely

$$r_t(s,a) = E(R_t | s_t = s, a_t = a).$$

3. Risk-sensitive criteria: The expected cumulative return is by far the most common goal for planning. However it is not the only one possible. For example, one may consider the following risk-sensitive return function:

$$J_{T,\lambda}^{\pi}(s) = \frac{1}{\lambda} \log E^{\pi,s}(\exp(\lambda \sum_{t=0}^{T} R_t)).$$

   For $\lambda > 0$, the exponent gives higher weight to high rewards, and the opposite for $\lambda < 0$.

4.3.2 Infinite Horizon Problems

We next consider planning problems that extend to an unlimited time horizon, $t = 0, 1, 2, \ldots$. Such planning problems arise when the system in question is expected to operate for a long time, or a large number of steps, possibly with no specific "closing" time. Infinite horizon problems are most often defined for stationary problems. In that case, they enjoy the important advantage that optimal policies can be found among the class of stationary policies. We will restrict attention here to stationary models. As before, we have the running reward function $r(s, a)$, which extends to all $t \geq 0$. The reward obtained at stage t is $R_t = r(s_t, a_t)$.

Discounted return: The most common performance criterion for infinite horizon problems is the expected discounted return:

$$J_\alpha^\pi(s) = E^\pi(\sum_{t=0}^{\infty} \alpha^t r(s_t, a_t) | s_0 = s) \equiv E^{\pi,s}(\sum_{t=0}^{\infty} \alpha^t r(s_t, a_t))$$

Here $0 < \alpha < 1$ is the discount factor. Mathematically, the discount factor ensures convergence of the sum (whenever the reward sequence is bounded). This make the problem "well behaved", and relatively easy to analyze.

Average return: Here we are interested to maximize the long-term average return. The most common definition of the long-term average return is

$$J_{av}^\pi(s) = \liminf_{T \to \infty} E^{\pi,s}(\frac{1}{T} \sum_{t=0}^{T-1} r(s_t, a_t))$$

The theory of average-return planning problems is more involved, and relies to a larger extent on the theory of Markov chains.

4.3.3 Stochastic Shortest-Path Problems

In an important class of planning problems, the time horizon is not set beforehand, but rather the problem continues until a certain event occurs. This event can be defined as reaching some goal state. Let $S_G \subset S$ define the set of *goal states*. Define

$$\tau = \inf\{t \geq 0 : s_t \in S_G\}$$

as the first time in which a goal state is reached. The total expected return for this problem is defined as:

$$J_{ssp}^\pi(s) = E^{\pi,s}(\sum_{t=0}^{\tau-1} r(s_t, a_t) + r_G(s_\tau))$$

Here $r_G(s),\ s \in S_G$ specified the reward at goal states.

This class of problems provides a natural extension of the standard shortest-path problem to stochastic settings. Some conditions on the system dynamics and reward function must be imposed for the problem to be well posed (e.g., that a goal state may be reached with probability one). Such problems are known as stochastic shortest path problems, or also episodic planning problems.

4.4 *Sufficiency of Markov Policies

In all the performance criteria defined above, the criterion is composed of sums of terms of the form $E(r_t(s_t, a_t))$. It follows that if two control policies induce the same marginal probability distributions $p_t(s_t, a_t)$ over the state-action pairs (s_t, a_t) for all $t \geq 0$, they will have the same performance.

Using this observation, the next claim implies that it is enough to consider the set of (randomized) Markov policies in the above planning problems.

Proposition 4.1. *Let $\pi \in \Pi_{GR}$ be a general (history-dependent, randomized) control policy. Let*

$$p_t^{\pi,s_0}(s,a) = P^{\pi,s_0}(s_t = s, a_t = a), \qquad (s,a) \in S_t \times A_t$$

Denote the marginal distributions induced by (s_t, a_t) on the state-action pairs (s_t, a_t), for all $t \geq 0$. Then there exists a randomized Markov policy $\tilde{\pi}$ that induces the same marginal probabilities (for all initial states s_0).

Exercise 4.2 (**Challenge Problem 1**)**.** Prove Proposition 4.1.

Note: If you consult a reference or a friend, mention that in your solution.

4.5 Finite-Horizon Dynamic Programming

Recall that we consider the expected total reward criterion, which we denote as

$$J^{\pi}(s_0) = E^{\pi,s_0}\left(\sum_{t=0}^{T-1} r_t(s_t, a_t) + r_T(s_T)\right)$$

Here π is the control policy used, and s_0 is a given initial state. We wish to maximize $J^{\pi}(s_0)$ over all control policies, and find an optimal policy π^* that achieves the maximal reward $J^*(s_0)$ for all initial states s_0. Thus,

$$J^*(s_0) \stackrel{\Delta}{=} J^{\pi *}(s_0) = \max_{\pi \in \Pi_{GR}} J^{\pi}(s_0)$$

4.5.1 The Principle of Optimality

The celebrated principle of optimality (stated by Bellman) applies to a large class of multi-stage optimization problems, and is at the heart of DP. As a general principle, it states that:

The tail of an optimal policy is optimal for the "tail" problem.

This principle is not an actual claim, but rather a guiding principle that can be applied in different ways to each problem. For example, considering our finite-horizon problem, let $\pi^* = (\pi_0, \ldots, \pi_{T-1})$ denote an optimal Markov policy. Take any state $s_t = s'$ which has a positive probability to be reached under π^*, namely $P^{\pi,s_0}(s_t = s') > 0$. Then the tail policy $(\pi_t, \ldots, \pi_{T-1})$ is optimal for the "tail" criterion $J^{\pi}_{t:T}(s') = E^{\pi}\left(\sum_{k=t}^{T} R_k | s_t = s'\right)$.

4.5.2 Dynamic Programming for Policy Evaluation

As a "warmup", let us evaluate the reward of a given policy. Let $\pi = (\pi_0, \ldots, \pi_{T-1})$ be a given Markov policy. Define the following reward-to-go function, or value function:

$$V^{\pi}_k(s) = E^{\pi}\left(\sum\nolimits_{t=k}^{T} R_t | s_k = s\right)$$

Observe that $V^{\pi}_0(s_0) = J^{\pi}(s_0)$.

Lemma 4.1 (**Value Iteration**). *$V^{\pi}_k(s)$ may be computed by the backward recursion:*

$$V^{\pi}_k(s) = \left\{ r_k(s,a) + \sum\nolimits_{s' \in S_{k+1}} p_k(s'|s,a)\, V^{\pi}_{k+1}(s') \right\}_{a=\pi_k(s)}, \quad s \in S_k$$

for $k = T-1, \ldots, 0$, starting with $V^{\pi}_T(s) = r_T(s)$.

Proof. Observe that:

$$\begin{aligned}
V^{\pi}_k(s) &= E^{\pi}\left(R_k + \sum\nolimits_{t=k+1}^{N} R_t |\ s_k = s, a_k = \pi_k(s)\right) \\
&= E^{\pi}\left(E^{\pi}\left(R_k + \sum\nolimits_{t=k+1}^{N} R_t |\ s_k = s, a_k = \pi_k(s), s_{k+1}\right) | s_k = s, a_k = \pi_k(s)\right) \\
&= E^{\pi}\left(r(s_k, a_k) + V^{\pi}_{k+1}(s_{k+1}) | s_k = s, a_k = \pi_k(s)\right) \\
&= r_k(s, \pi_k(s)) + \sum\nolimits_{s' \in S_{k+1}} p_k(s'|s, \pi_k(s))\, V^{\pi}_{k+1}(s')
\end{aligned}$$

□

Remarks:

- Note that $\sum_{s' \in S_{k+1}} p_k(s'|s,a)\, V_{k+1}^{\pi}(s') = E^{\pi}(V_{k+1}^{\pi}(s_{k+1})|s_k = s, a_k = a)$.
- For the more general reward function $\tilde{r}_t(s,a,s')$, the recursion takes the form

$$V_k^{\pi}(s) = \left\{ \sum\nolimits_{s' \in S_{k+1}} p_k(s'|s,a)[r_k(s,a,s') + V_{k+1}^{\pi}(s')] \right\}_{a=\pi_k(s)}$$

  A similar observation applies to all Dynamic Programming equations below.

4.5.3 Dynamic Programming for Policy Optimization

We next define the optimal value function at each time $k \geq 0$:

$$V_k(s) = \max_{\pi^k} E^{\pi^k} \left(\sum\nolimits_{t=k}^{T} R_t | s_k = s \right), \quad s \in S_k$$

The maximum is taken over "tail" policies $\pi^k = (\pi_k, \ldots, \pi_{T-1})$ that start from time k. Note that π^k is allowed to be a general policy, i.e., history-dependent and randomized. Obviously, $V_0(s_0) = J^*(s_0)$.

Theorem 4.3 (**Finite-horizon Dynamic Programming**). *The following holds:*

1. *Backward recursion: Set $V_T(s) = r_T(s)$ for $s \in S_T$.*
   For $k = T-1, \ldots, 0$, $V_k(s)$ may be computed using the following recursion:

$$V_k(s) = \max_{a \in A_k} \left\{ r_k(s,a) + \sum\nolimits_{s' \in S_{k+1}} p_k(s'|s,a)\, V_{k+1}(s') \right\}, \quad s \in S_k.$$

2. *Optimal policy: Any Markov policy π^* that satisfies, for $t = 0, \ldots, T-1$,*

$$\pi_t^*(s) \in \arg\max_{a \in A_t} \left\{ r_t(s,a) + \sum\nolimits_{s' \in S_{t+1}} p_t(s'|s,a)\, V_{t+1}(s') \right\}, \quad s \in S_t,$$

   is an optimal control policy. Furthermore, π^* maximizes $J^{\pi}(s_0)$ simultaneously for every initial state $s_0 \in S_0$.

Note that Theorem 4.3 specifies an optimal control policy which is a deterministic Markov policy.

Proof. **Part (i):**

We use induction to show that the stated backward recursion indeed yields the optimal value function. The idea is simple, but some care is needed with the notation since we

consider general policies, and not just Markov policies. The equality $V_T(s) = r_T(s)$ follows directly from the definition of V_T.

We proceed by backward induction. Suppose that $V_{k+1}(s)$ is the optimal value function for time $k+1$. We need to show that $V_k(s) = W_k(s)$, where

$$W_k(s) \stackrel{\Delta}{=} \max_{a \in A_k} \left\{ r_k(s,a) + \sum\nolimits_{s' \in S_{k+1}} p_k(s'|s,a)\, V_{k+1}(s') \right\}.$$

We will first establish that $V_k(s) \geq W_k(s)$, and then that $V_k(s) \leq W_k(s)$.

(a) We first show that $V_k(s) \geq W_k(s)$. For that purpose, it is enough to find a policy π^k so that $V_k^{\pi^k}(s) = W_k(s)$.

Fix $s \in S_k$, and define π^k as follows: Choose $a_k = \bar{a}$, where

$$\bar{a} \in \arg\max_{a \in A_k} \left\{ r_k(s,a) + \sum\nolimits_{s' \in S_{k+1}} p_k(s'|s,a)\, V_{k+1}(s') \right\},$$

and then, after observing $s_{k+1} = s'$, proceed with the optimal tail policy $\pi^{k+1}(s')$ that obtains $V_{k+1}^{\pi^{k+1}(s')}(s') = V_{k+1}(s')$. Proceeding similarly to Subsection 4.5.2 above (value iteration for a fixed policy), we obtain:

$$V_k^{\pi^k}(s) = r_k(s,\bar{a}) + \sum\nolimits_{s' \in S_{k+1}} p(s'|s,\bar{a})\, V_{k+1}^{\pi^{k+1}(s')}(s') \tag{4.2}$$

$$= r_k(s,\bar{a}) + \sum\nolimits_{s' \in S_{k+1}} p(s'|s,\bar{a})\, V_{k+1}(s') = W_k(s), \tag{4.3}$$

as was required.

(b) To establish $V_k(s) \leq W_k(s)$, it is enough to show that $V_k^{\pi^k}(s) \leq W_k(s)$ for any (general, randomized) "tail" policy π^k.

Fix $s \in S_k$. Consider then some tail policy $\pi^k = (\pi_k, \ldots \pi_{T-1})$. Note that this means that $a_t \sim \pi_t(a|h_{k:t})$, where $h_{k:t} = (s_k, a_k, s_{k+1}, a_{k+1}, \ldots, s_t)$. For each state-action pair $s \in S_k$ and $a \in A_k$, let $(\pi^k|s,a)$ denote the tail policy π^{k+1} from time $k+1$ onwards which is obtained from π^k given that $s_k = s,\ a_k = a$. As before, by value iteration for a fixed policy,

$$V_k^{\pi^k}(s) = \sum\nolimits_{a \in A_k} \pi_k(a|s) \left\{ r_k(s,a) + \sum\nolimits_{s' \in S_{k+1}} p_k(s'|s,a)\, V_{k+1}^{(\pi^k|s,a)}(s') \right\}.$$

But since V_{k+1} is optimal,

$$V_k^{\pi^k}(s) \leq \sum\nolimits_{a \in A_k} \pi_k(a|s) \left\{ r_k(s,a) + \sum\nolimits_{s' \in S_{k+1}} p_k(s'|s,a)\, V_{k+1}(s') \right\}$$

$$\leq \max_{a \in A_k} \left\{ r_k(s,a) + \sum\nolimits_{s' \in S_{k+1}} p_k(s'|s,a)\, V_{k+1}(s') \right\} = W_k(s),$$

which is the required inequality in (b).

Part (ii) (Outline - **exercise**):

Let π^* be the (Markov) policy defined in part 2 of Theorem 4.3. Using value iteration for this policy, prove by backward induction that $V_k^{\pi^*} = V_k$. □

To summarize:

- The optimal value function can be computed by backward recursion. This recursive equation is known as the *dynamic programming equation*, *optimality equation*, or *Bellman's Equation*.
- Computation of the value function in this way is known as the *finite-horizon value iteration* algorithm.
- The value function is computed for all states at each stage.
- An optimal policy is easily derived from the optimal value.
- The optimization in each stage is performed in the action space. The total number of minimization operations needed is $T \times |S|$ - each over $|A|$ choices. This replaces "brute force" optimization in policy space, with tremendous computational savings as the number of Markov policies is $|A|^{T \times |S|}$.

4.5.4 The Q function

Let

$$Q_k(s,a) \stackrel{\Delta}{=} r_k(s,a) + \sum\nolimits_{s' \in S_k} p_k(s'|s,a)\, V_k(s').$$

This is known as the optimal state-action value function, or simply as the *Q-function*. $Q_k(s,a)$ is the expected return from stage k onward, if we choose $a_k = a$ and then proceed optimally.

Theorem 4.3 can now be succinctly expressed as

$$V_k(s) = \max_{a \in A_k} Q_k(s,a),$$

and

$$\pi_k^*(s) \in \arg\max_{a \in A_k} Q_k(s,a).$$

The Q function provides the basis for the Q-learning algorithm, which is one of the basic Reinforcement Learning algorithms.

4.6 Exercises

Exercise 4.3 (**Markov chains**). Let $\{X_n\}$ be a time-homogenous Markov processes in discrete time which takes values in $\{0, 1, ..\}$ (an infinite countable set).

1. Assume the process satisfies for each $i \in \{0, 1, ...\}$:

$$p_{i,0} = q, \quad p_{i,i+1} = 1 - q, \quad 0 < q < 1.$$

   Plot the state transition diagram for $\{0, 1, 2, 3\}$. If the chain is recurrent, find the stationary distribution, if it is not find the transient states.

2. Consider the same process as above and assume $P(X_0 = 0) = 1$. Define $Y_n = |\{\tau : X_\tau = 0, \tau \leq n\}|$ as the number of visits to 0 until time n. Also define $Z_n = \begin{pmatrix} X_n & Y_n \end{pmatrix}^T$. Is $\{Z_n\}$ a Markov process? Is it recurrent? Is it transient?

3. Assume the process satisfies for each $i \in \{1, 2, ...\}$:

$$p_{0,1} = 1, \quad p_{i,i+1} = p_{i,i-1} = 0.5.$$

   Is the process recurrent? If so, find a stationary distribution if it exists or explain why there is none. If the process is not recurrent, what are the transient states?

Chapter 5

MDPs with Discounted Return

This lecture covers the basic theory and main solution methods for stationary MDPs over an infinite horizon, with the discounted return criterion. In this case, stationary policies are optimal.

The discounted return problem is the most "well behaved" among all infinite horizon problems (such as average return and stochastic shortest path), and the theory of it is relatively simple, both in the planning and the learning contexts. For that reason, as well as its usefulness, we will consider here the discounted problem and its solution in some detail.

5.1 Problem Statement

We consider a stationary (time-invariant) MDP, with a finite state space S, finite action set A, and transition kernel $P = (P(s'|s,a))$ over the infinite time horizon $\mathbf{T} = \{0, 1, 2, \ldots\}$.

Our goal is to maximize the expected discounted return, which is defined for each control policy π and initial state $s_0 = s$ as follows:

$$\begin{aligned} J_\gamma^\pi(s) &= E^\pi(\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) | s_0 = s) \\ &\equiv E^{\pi,s}(\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)) \end{aligned}$$

Here,

- $r : S \times A \to \mathbb{R}$ is the (running, or instantaneous) reward function.
- $\gamma \in (0, 1)$ is the discount factor.

We observe that $\gamma < 1$ ensures convergence of the infinite sum (since that the rewards $r(s_t, a_t)$ are uniformly bounded). With $\gamma = 1$ we obtain the total return criterion, which is harder to handle due to possible divergence of the sum.

Let $J^*_\gamma(s)$ denote the maximal value of the discounted return, over all (possibly history dependent and randomized) control policies, i.e.,

$$J^*_\gamma(s) = \sup_{\pi \in \Pi_{GR}} J^\pi_\gamma(s).$$

Our goal is to find an optimal control policy π^* that attains that maximum (for all initial states), and compute the numeric value of the optimal return $J^*_\gamma(s)$. As we shall see, for this problem there always exists an optimal policy which is a (deterministic) stationary policy.

Note: As usual, the discounted performance criterion can be defined in terms of cost:

$$J^\pi_\gamma(s) = E^{\pi,s}(\sum_{t=0}^{\infty} \gamma^t c(s_t, a_t))$$

where $c(s, a)$ is the running cost function. Our goal is then to minimize the discounted cost $J^\pi_\gamma(s)$.

5.2 The Fixed-Policy Value Function

We start the analysis by defining and computing the value function for a fixed stationary policy. This intermediate step is required for later analysis of our optimization problem, and also serves as a gentle introduction to the value iteration approach.

For a stationary policy $\pi : S \to A$, we define the value function $V^\pi(s),\ s \in S$ simply as the corresponding discounted return:

$$V^\pi(s) \stackrel{\Delta}{=} E^{\pi,s}\left(\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right) = J^\pi_\gamma(s), \quad s \in S$$

Lemma 5.1. *V^π satisfies the following set of $|S|$ linear equations:*

$$V^\pi(s) = r(s, \pi(s)) + \gamma \sum\nolimits_{s' \in S} p(s'|s, \pi(s)) V^\pi(s')\ , \quad s \in S. \tag{5.1}$$

Proof. We first note that

$$\begin{aligned} V^\pi(s) &\stackrel{\Delta}{=} E^\pi(\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) | s_0 = s) \\ &= E^\pi(\sum_{t=1}^{\infty} \gamma^{t-1} r(s_t, a_t) | s_1 = s), \end{aligned}$$

since both the model and the policy are stationary. Now,

$$\begin{aligned}
V^\pi(s) &= r(s,\pi(s)) + E^\pi(\sum_{t=1}^{\infty} \gamma^t r(s_t,\pi(s_t))|s_0 = s) \\
&= r(s,\pi(s)) + E^\pi\left[E^\pi\left(\sum_{t=1}^{\infty} \gamma^t r(s_t,\pi(s_t))|s_0 = s, s_1 = s'\right) \middle| s_0 = s\right] \\
&= r(s,\pi(s)) + \sum_{s'\in S} p(s'|s,\pi(s)) E^\pi(\sum_{t=1}^{\infty} \gamma^t r(s_t,\pi(s_t))|s_1 = s') \\
&= r(s,\pi(s)) + \gamma \sum_{s'\in S} p(s'|s,\pi(s)) E^\pi(\sum_{t=1}^{\infty} \gamma^{t-1} r(s_t,a_t)|s_1 = s') \\
&= r(s,\pi(s)) + \gamma \sum_{s'\in S} p(s'|s,\pi(s)) V^\pi(s').
\end{aligned}$$

The first equality is by the smoothing theorem. The second equality follows since $s_0 = s$ and $a_t = \pi(s_t)$, the third equality follows similarly to the finite-horizon case (Lemma 4.1 in the previous lecture), the fourth is simple algebra, and the last by the observation above. □

We can write the linear equations in (5.1) in vector form as follows. Define the column vector $r^\pi = (r^\pi(s))_{s\in S}$ with components $r^\pi(s) = r(s,\pi(s))$, and the transition matrix P^π with components $P^\pi(s'|s) = p(s'|s,\pi(s))$. Finally, let V^π denote a column vector with components $V^\pi(s)$. Then (5.1) is equivalent to the linear equation set

$$V^\pi = r^\pi + \gamma P^\pi V^\pi \tag{5.2}$$

Lemma 5.2. *The set of linear equations* (5.1) *or* (5.2)*, with V^π as variables, has a unique solution V^π, which is given by*

$$V^\pi = (I - \gamma P^\pi)^{-1} r^\pi.$$

Proof. We only need to show that the square matrix $I - \gamma P^\pi$ is non-singular. Let (λ_i) denote the eigenvalues of the matrix P^π. Since P^π is a stochastic matrix (row sums are 1), then $|\lambda_i| \le 1$. Now, the eignevalues of $I - \gamma P^\pi$ are $(1 - \gamma\lambda_i)$, and satisfy $|1 - \gamma\lambda_i| \ge 1 - \gamma > 0$. □

Combining Lemma 5.1 and Lemma 5.2, we obtain

Proposition 5.1. *The fixed-policy value function $V^\pi = [V^\pi(s)]$ is the unique solution of equation* (5.2)*, given by*

$$V^\pi = (I - \gamma P^\pi)^{-1} r^\pi.$$

Proposition 5.1 provides a closed-form formula for computing V^π. For large systems, computing the inverse $(I - \gamma P^\pi)^{-1}$ may be hard. In that case, the following value iteration algorithm provides an alternative, iterative method for computing V^π.

Algorithm 5.1. ***Fixed-policy value iteration***

1. *Let $V_0 = (V_0(s))_{s\in S}$ be arbitrary.*
2. *For $n = 0, 1, 2, \ldots$, set*

$$V_{n+1}(s) = r(s, \pi(s)) + \gamma \sum\nolimits_{s'\in S} p(s'|s, \pi(s))V_n(s') \,, \qquad s \in S$$

or, equivalently,

$$V_{n+1} = r^\pi + \gamma P^\pi V_n.$$

Proposition 5.2 (**Convergence of fixed-policy value iteration**). *We have $V_n \to V^\pi$ component-wise, that is,*

$$\lim_{n\to\infty} V_n(s) = V^\pi(s), \qquad s \in S.$$

Proof. Note first that

$$\begin{aligned} V_1(s) &= r(s, \pi(s)) + \gamma \sum\nolimits_{s'\in S} p(s'|s, \pi(s))V_0(s') \\ &= E^\pi(r(s_0, a_0) + \gamma V_0(s_1)|s_0 = s). \end{aligned}$$

Continuing similarly, we obtain that

$$V_n(s) = E^\pi(\sum_{t=0}^{n-1} \gamma^t r(s_t, a_t) + \gamma^n V_0(s_n)|s_0 = s).$$

Note that $V_n(s)$ is the n-stage discounted return, with terminal reward $r_n(s_n) = V_0(s_n)$. Comparing with the definition of V^π, we can see that

$$V^\pi(s) - V_n(s) = E^\pi(\sum_{t=n}^{\infty} \gamma^t r(s_t, a_t) - \gamma^n V_0(s_n)|s_0 = s).$$

Denoting $R_{\max} = \max_{s,a}|r(s, a)|$, $\bar{V}_0 = \max_s|V(s)|$ we obtain

$$|V^\pi(s) - V_n(s)| \le \gamma^n(\frac{R_{\max}}{1-\gamma} + \bar{V}_0)$$

which converges to 0 since $\gamma < 1$. □

Comments:

- The proof provides an explicit bound on $|V^\pi(s) - V_n(s)|$. It may be seen that the convergence is exponential, with rate $O(\gamma^n)$.
- Using vector notation, it may be seen that

$$V_n = r^\pi + P^\pi r^\pi + \ldots + (P^\pi)^{n-1} r^\pi + (P^\pi)^n V_0 = \sum_{t=0}^{n-1} (P^\pi)^t r^\pi + (P^\pi)^n V_0.$$

  Similarly, $V^\pi = \sum_{t=0}^{\infty} (P^\pi)^t r^\pi$.

In summary:

- Proposition 5.1 allows to compute V^π by solving a set of $|S|$ linear equations.
- Proposition 5.2 computes V^π by an infinite recursion, that converges exponentially fast.

5.3 Overview: The Main DP Algorithms

We now return to the optimal planning problem defined in Section 5.1. Recall that $J^*_\gamma(s) = \sup_{\pi \in \Pi_{GR}} J^\pi_\gamma(s)$ is the optimal discounted return. We further denote

$$V^*(s) \stackrel{\Delta}{=} J^*_\gamma(s), \quad s \in S,$$

and refer to V^* as the optimal value function. Depending on the context, we consider V^* either as a function $V^* : S \to \mathbb{R}$, or as a column vector $V^* = [V(s)]_{s \in S}$.

The following optimality equation provides an explicit characterization of the value function, and shows that an optimal stationary policy can easily be computed if the value function is known.

Theorem 5.1 (**Bellman's Optimality Equation**). *The following statements hold:*

1. *V^* is the unique solution of the following set of (nonlinear) equations:*

$$V(s) = \max_{a \in A} \left\{ r(s,a) + \gamma \sum\nolimits_{s' \in S} p(s'|s,a) V(s') \right\}, \quad s \in S. \tag{5.3}$$

2. *Any stationary policy π^* that satisfies*

$$\pi^*(s) \in \arg\max_{a \in A} \left\{ r(s,a) + \gamma \sum\nolimits_{s' \in S} p(s'|s,a) V(s') \right\},$$

   is an optimal policy (for any initial state $s_0 \in S$).

The optimality equation (5.3) is non-linear, and generally requires iterative algorithms for its solution. The main iterative algorithms are **value iteration** and **policy iteration**.

Algorithm 5.2. ***Value iteration***

1. *Let* $V_0 = (V_0(s))_{s \in S}$ *be arbitrary.*
2. *For* $n = 0, 1, 2, \ldots$, *set*

$$V_{n+1}(s) = \max_{a \in A} \left\{ r(s,a) + \gamma \sum\nolimits_{s' \in S} p(s'|s,a) V_n(s') \right\}, \quad \forall s \in S$$

Theorem 5.2 (**Convergence of value iteration**). *We have* $\lim_{n \to \infty} V_n = V^*$ *(component-wise). The rate of convergence is exponential, at rate* $O(\gamma^n)$.

The value iteration algorithm iterates over the value functions, with asymptotic convergence. The policy iteration algorithm iterates over stationary policies, with each new policy better than the previous one. This algorithm converges to the optimal policy in a finite number of steps.

Algorithm 5.3. ***Policy iteration***

1. *Initialization: choose some stationary policy* π_0.
2. *For* $k = 0, 1, \ldots$:
   (a) *Policy evaluation: compute* V^{π_k}. *% For example, use the explicit formula* $V^{\pi_k} = (I - \gamma P^{\pi_k})^{-1} r^{\pi_k}$.
   (b) *Policy Improvement: Compute* π_{k+1}, *a greedy policy with respect to* V^{π_k}:

$$\pi_{k+1}(s) \in \arg\max_{a \in A} \left\{ r(s,a) + \gamma \sum\nolimits_{s' \in S} p(s'|s,a) V^{\pi_k}(s') \right\}, \quad \forall s \in S.$$

   (c) *Stop if* $V^{\pi_{k+1}} = V^{\pi_k}$ *(or if* V^{π_k} *satisfied the optimality equation), else continue.*

Theorem 5.3 (**Convergence of policy iteration**). *The following statements hold:*

1. *Each policy* π_{k+1} *is improving over the previous one* π_k, *in the sense that* $V^{\pi_{k+1}} \geq V^{\pi_k}$ *(component-wise).*
2. $V^{\pi_{k+1}} = V^{\pi_k}$ *if and only if* π_k *is an optimal policy.*
3. *Consequently, since the number of stationary policies is finite,* π_k *converges to the optimal policy after a finite number of steps.*

An additional solution method for DP planning relies on a Linear Programming formulation of the problem. A Linear Program (LP) is simply an optimization problem with linear objective function and linear constraints. We will provide additional details later in this Lecture.

5.4 Contraction Operators

The basic proof methods of the DP results mentioned above rely on the concept of a *contraction operator*. We provide here the relevant mathematical background, and illustrate the contraction properties of some basic Dynamic Programming operators.

5.4.1 The contraction property

Recall that a norm $||\cdot||$ over $\mathbb{R}^n$ is a real-valued function $\|\cdot\| : \mathbb{R}^n \to \mathbb{R}$ such that, for any pair of vectors $x, y \in \mathbb{R}^n$ and scalar a,

1. $||ax|| = |a| \cdot ||ax||$,
2. $||x + y|| \leq ||x|| + ||y||$,
3. $||x|| = 0$ only if $x = 0$.

Common examples are the p-norm $||x||_p = (\sum_{i=1}^n |x_i|^p)^{1/p}$ for $p \geq 1$, and in particular the Euclidean norm ($p = 2$). Here we will mostly use the max-norm:

$$||x||_\infty = \max_{1 \leq i \leq n} |x_i|.$$

Let $T : \mathbb{R}^d \to \mathbb{R}^d$ be a vector-valued function over $\mathbb{R}^d$ ($d \geq 1$). We equip $\mathbb{R}^d$ with some norm $||\cdot||$, and refer to T as an *operator* over $\mathbb{R}^d$. Thus, $T(v) \in \mathbb{R}^d$ for any $v \in \mathbb{R}^d$. We also denote $T^n(v) = T(T^{n-1}(v))$ for $n \geq 2$. For example, $T^2(v) = T(T(v))$.

Definition 5.1. *The operator T is called a contraction operator if there exists $\beta \in (0, 1)$ (the contraction coefficient) such that*

$$||T(v_1) - T(v_2)|| \leq \beta ||v_1 - v_2||,$$

for all $v_1, v_2 \in \mathbb{R}^d$

5.4.2 The Banach Fixed Point Theorem

The following celebrated result applies to contraction operators. While we quote the result for $\mathbb{R}^d$, we note that it applies in much greater generality to any Banach space (a complete normed space), or even to any complete metric space, with essentially the same proof.

Theorem 5.4 (Banach's fixed point theorem). *Let $T : \mathbb{R}^d \to \mathbb{R}^d$ be a contraction operator. Then*

1. *The equation $T(v) = v$ has a unique solution $V^* \in \mathbb{R}^d$.*
2. *For any $v_0 \in \mathbb{R}^d$, $\lim_{n\to\infty} T^n(v_0) = V^*$. In fact, $||T^n(v_0) - V^*|| \leq O(\beta^n)$, where β is the contraction coefficient.*

Proof. (outline)

1. Uniqueness: Let V_1 and V_2 be two solutions of $T(v) = v$, then

$$||V_1 - V_2|| = ||T(V_1) - T(V_2)|| \leq \beta||V_1 - V_2||,$$

which implies that $||V_1 - V_2|| = 0$, hence $V_1 = V_2$.

Existence (outline): (i) show that $V_n \stackrel{\Delta}{=} T^n(V_0)$ (with V_0 arbitrary) is a Cauchy sequence. (ii) Since any Cauchy sequence in $\mathbb{R}^d$ converges, this implies that V_n converges to some $V^* \in \mathbb{R}^d$. (iii) Now show that V^* satisfies the equation $T(v) = v$.

2. We have just shown that, for any V_0, $V_n \stackrel{\Delta}{=} T^n(V_0)$ converges to a solution of $T(v) = v$, and that solution was shown before to be unique. Furthermore, we have

$$\begin{aligned} ||V_n - V^*|| &= ||T(V_{n-1}) - T(V^*)|| \\ &\leq \beta||V_{n-1} - V^*|| \leq \ldots \leq \beta^n||V_0 - V^*|| \end{aligned}$$

□

5.4.3 The Dynamic Programming Operators

We next define the basic Dynamic Programming operators, and show that they are in fact contraction operators.

For a fixed stationary policy $\pi : S \to A$, define the fixed policy DP operator $T^\pi : \mathbb{R}^{|S|} \to \mathbb{R}^{|S|}$ as follows: For any $V = (V(s)) \in R^{|S|}$,

$$(T^\pi(V))(s) = r(s, a) + \gamma \sum\nolimits_{s' \in S} p(s'|s, \pi(s))V(s'), \quad s \in S$$

In our column-vector notation, this is equivalent to $T^\pi(V) = r^\pi + \gamma P^\pi V$.

Similarly, define the discounted-return **Dynamic Programming Operator** $T^* : \mathbb{R}^{|S|} \to \mathbb{R}^{|S|}$ as follows: For any $V = (V(s)) \in R^{|S|}$,

$$(T^*(V))(s) = \max_{a \in A} \left\{ r(s, a) + \gamma \sum\nolimits_{s' \in S} p(s'|s, a)V(s') \right\}, \quad s \in S$$

We note that T^π is a linear operator, while T^* is generally non-linear due to the maximum operation.

Let $||V||_\infty \stackrel{\Delta}{=} \max_{s \in S}|V(s)|$ denote the max-norm of V. Recall that $0 < \gamma < 1$.

Theorem 5.5 (**Contraction property**). *The following statements hold:*

1. *T^π is a γ-contraction operator with respect to the max-norm, namely $||T^\pi(V_1) - T^\pi(V_2)||_\infty \leq \gamma||V_1 - V_2||_\infty$ for all $V_1, V_2 \in R^{|S|}$.*

2. Similarly, T^* is an γ-contraction operator with respect to the max-norm.

Proof. 1. Fix V_1, V_2. For every state s,

$$\begin{aligned}|T^\pi(V_1)(s) - T^\pi(V_2)(s)| &= \left|\gamma \sum\nolimits_{s'} p(s'|s,\pi(s))[V_1(s') - V_2(s')]\right| \\ &\le \gamma \sum\nolimits_{s'} p(s'|s,\pi(s)) \left|V_1(s') - V_2(s')\right| \\ &\le \gamma \sum\nolimits_{s'} p(s'|s,\pi(s)) \|V_1 - V_2\|_\infty = \gamma \|V_1 - V_2\|_\infty .\end{aligned}$$

Since this holds for every $s \in S$ the required inequality follows.

2. The proof here is more intricate due to the maximum operation. As before, we need to show that $|T^*(V_1)(s) - T^*(V_2)(s)| \le \gamma \|V_1 - V_2\|_\infty$. Fixing the state s, we consider separately the positive and negative parts of the absolute value:

(a) $T^*(V_1)(s) - T^*(V_2)(s) \le \gamma \|V_1 - V_2\|_\infty$: Let $\bar{a}$ denote an action that attains the maximum in $T^*(V_1)(s)$, namely $\bar{a} \in \arg\max_{a \in A} \left\{ r(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V_1(s') \right\}$. Then

$$\begin{aligned}T^*(V_1)(s) &= r(s,\bar{a}) + \gamma \sum\nolimits_{s' \in S} p(s'|s,\bar{a}) V_1(s') \\ T^*(V_2)(s) &\ge r(s,\bar{a}) + \gamma \sum\nolimits_{s' \in S} p(s'|s,\bar{a}) V(s')\end{aligned}$$

Since the same action $\bar{a}$ appears in both expressions, we can now continue to show the inequality (a) similarly to 1.

(b) $T^*(V_2)(s) - T^*(V_1)(s) \le \gamma \|V_1 - V_2\|_\infty$: Follows symmetrically to (a).

The inequalities (a) and (b) together imply that $|T^*(V_1)(s) - T^*(V_2)(s)| \le \gamma \|V_1 - V_2\|_\infty$. Since this holds for any state s, it follows that $||T^*(V_1) - T^*(V_2)||_\infty \le \gamma \|V_1 - V_2\|_\infty$. □

5.5 Proof of Bellman's Optimality Equation

We prove in this section Theorem 5.1, which is restated here:

Theorem (Bellman's Optimality Equation)**.** *The following statements hold:*

1. V^* is the unique solution of the following set of (nonlinear) equations:

$$V(s) = \max_{a \in A} \left\{ r(s,a) + \gamma \sum\nolimits_{s' \in S} p(s'|s,a) V(s') \right\}, \quad s \in S. \tag{5.4}$$

2. Any stationary policy π^* that satisfies

$$\pi^*(s) \in \arg\max_{a \in A} \left\{ r(s,a) + \gamma \sum\nolimits_{s' \in S} p(s'|s,a) V(s') \right\},$$

is an optimal policy (for any initial state $s_0 \in S$).

We observe that the Optimality equation in part 1 is equivalent to $V = T^*(V)$ where T^* is the optimal DP operator from the previous section, which was shown to be a contraction operator with coefficient γ. The proof also uses the value iteration property of Theorem 5.2, which is proved in the next section.

Proof of Theorem 5.1: We prove each part.

1. As T^* is a contraction operator, existence and uniqueness of the solution to $V = T^*(V)$ follows from the Banach fixed point theorem. Let $\hat{V}$ denote that solution. It also follows by that theorem that $(T^*)^n(V_0) \to \hat{V}$ for any V_0. But in Theorem 5.2 we show that $(T^*)^n(V_0) \to V^*$, hence $\hat{V} = V^*$, so that V^* is indeed the unique solution of $V = T^*(V)$.

2. By definition of π^* we have

$$T^{\pi*}(V^*) = T^*(V^*) = V^*,$$

   where the last equality follows from part 1. Thus the optimal value function satisfied the equation $T^{\pi*}V^* = V^*$. But we already know (from Prop. 5.2) that $V^{\pi*}$ is the unique solution of that equation, hence $V^{\pi*} = V^*$. This implies that π^* achieves the optimal value (for any initial state), and is therefore an optimal policy as stated.

□

5.6 Value Iteration

The value iteration algorithm allows to compute the optimal value function V^* iteratively to any required accuracy. The Value Iteration algorithm (Algorithm 5.2) can be stated as follows:

1. Start with any initial value function $V_0 = (V_0(s))$.

2. Compute recursively, for $n = 0, 1, 2, \ldots$,

$$V_{n+1}(s) = \max_{a \in A} \sum\nolimits_{s' \in S} p(s'|s,a)[r(s,a,s') + \gamma V_n(s')], \quad \forall s \in S.$$

3. Apply a stopping rule obtain a required accuracy (see below).

In terms of the DP operator T^*, value iteration is simply stated as:

$$V_{n+1} = T^*(V_n), \quad n \geq 0.$$

Note that the number of operations for each iteration is $O(|A| \cdot |S|^2)$. Theorem 5.2 states that $V_n \to V^*$, exponentially fast. The proof follows.

Proof of Theorem 5.2: Using our previous results on value iteration for the finite-horizon problem, it follows that

$$V_n(s) = \max_{\pi} E^{\pi,s}(\sum_{t=0}^{n-1} \gamma^t R_t + \gamma^n V_0(s_n)).$$

Comparing to the optimal value function

$$V^*(s) = \max_{\pi} E^{\pi,s}(\sum_{t=0}^{\infty} \gamma^t R_t),$$

it may be seen that that

$$|V_n(s) - V^*(s)| \leq \gamma^n (\frac{R_{\max}}{1-\gamma} + ||V_0||_\infty).$$

As $\gamma < 1$, this implies that V_n converges to V^*_γ exponentially fast. □

5.6.1 Error bounds and stopping rules:

It is important to have an on-line criterion for the accuracy of the n-the step solution V_n. The exponential bound in the last theorem is usually too crude and can be improved. Since our goal is to find an optimal policy, we need to know how errors in V^* affect the sub-optimality of the derived policy. We quote some useful bounds without proof. More refined error bounds can be found in the texts on Dynamic Programming.

1. The distance of V_n from the optimal solution is upper bounded as follows:

   $$||V_n - V^*||_\infty \leq \tfrac{\gamma}{1-\gamma} ||V_n - V_{n-1}||_\infty$$

   Note that the right-hand side depends only on the computed values on the last two rounds, and hence is easy to apply. As the bound also decays exponentially (with rate γ), it allows to compute the value function to within any required accuracy. In particular, to ensure $||V_n - V^*||_\infty \leq \varepsilon$, we can use the stopping rule $||V_n - V_{n-1}||_\infty \leq \frac{1-\gamma}{\gamma}\varepsilon$.

2. If $||V - V^*||_\infty \leq \varepsilon$, then any stationary policy π that is greedy with respect to V, i.e., satisfies

   $$\pi(s) \in \arg\max_{a \in A} \left\{ r(s,a) + \gamma \sum\nolimits_{s' \in S} p(s'|s,a) V_n(s') \right\},$$

   is 2ε-optimal, namely $||V^\pi - V^*||_\infty \leq 2\varepsilon$. This allows obtain a 2ε-optimal policy from an ε-approximation of V^*.

5.7 Policy Iteration

The policy iteration algorithm, introduced by Howard (1960), computes an optimal policy π^* in a finite number of steps. This number is typically small (on the same order as $|S|$).

The basic principle behind Policy Iteration is Policy Improvement. Let π be a stationary policy, and let V^π denote its value function. A stationary policy $\bar{\pi}$ is called π- improving if it is a greedy policy with respect to V^π, namely

$$\bar{\pi}(s) \in \arg\max_{a \in A} \left\{ r(s,a) + \gamma \sum\nolimits_{s' \in S} p(s'|s,a) V^\pi(s') \right\}, \quad s \in S.$$

Lemma 5.3 (Policy Improvement)**.** *We have $V^{\bar{\pi}} \geq V^\pi$ (component-wise), and equality holds if and only if π is an optimal policy.*

Proof. Observe first that

$$V^\pi = T^\pi(V^\pi) \leq T^*(V^\pi) = T^{\bar{\pi}}(V^\pi)$$

The first equality follows since V^π is the value function for the policy π, the inequality follows because of the maximization in the definition of T^*, and the last equality by definition of the improving policy $\bar{\pi}$.

It is easily seen that T^π is a monotone operator (for any policy π), namely $V_1 \leq V_2$ implies $T^\pi V_1 \leq T^\pi V_2$. Applying $T_\gamma^{\bar{\pi}}$ repeatedly to both sides of the above inequality $V^\pi \leq T^{\bar{\pi}} V^\pi$ therefore gives

$$V^\pi \leq T^{\bar{\pi}}(V^\pi) \leq (T^{\bar{\pi}})^2(V^\pi) \leq \cdots \leq \lim_{n \to \infty} (T^{\bar{\pi}})^n (V^\pi) = V^{\bar{\pi}},$$

where the last equality follows by value iteration. This establishes the first claim. The equality claim is left as an **exercise**. □

The policy iteration algorithm performs successive rounds of policy improvement, where each policy π_{k+1} improves the previous one π_k. Since the number of stationary policies is bounded, so is the number of strict improvements, and the algorithm must terminate with an optimal policy after a finite number of steps.

In terms of computational complexity, Policy Iteration requires $O(|A| \cdot |S|^2 + |S|^3)$ operations per step, with the number of steps being typically small. In contrast, Value Iteration requires $O(|A| \cdot |S|^2)$ per step, but the number of required iterations may be large, especially when the discount factor γ is close to 1.

5.8 Some Variants on Value Iteration and Policy Iteration

5.8.1 Value Iteration - Gauss Seidel Iteration

In the standard value iteration: $V_{n+1} = T^*(V_n)$, the vector V_n is held fixed while all entries of V_{n+1} are updated. An alternative is to update each element $V_n(s)$ of that vector as

to $V_{n+1}(s)$ as soon as the latter is computed, and continue the calculation with the new value. This procedure is guaranteed to be "as good" as the standard one, in some sense, and often speeds up convergence.

5.8.2 Asynchronous Value Iteration

Here, in each iteration $V_n \Rightarrow V_{n+1}$, only a subset of the entries of V_n (namely, a subset of all states) is updated. It can be shown that if each state is updated infinitely often, then $V_n \to V^*$. Asynchronous update can be used to focus the computational effort on "important" parts of a large-state space.

5.8.3 Modified (a.k.a. Generalized or Optimistic) Policy Iteration

This scheme combines policy improvement steps with value iteration for policy evaluation. This way the requirement for exact policy evaluation (computing $V^{\pi_k} = (I - \gamma P^{\pi_k})^{-1} r^{\pi_k}$) is avoided.

The procedure starts with some initial value vector V_0, and iterates as follows:

- Greedy policy computation:

  Compute $\pi_k \in \arg\max_\pi T^\pi(V_k)$, a greedy policy with respect to V_k.

- Partial value iteration:

  Perform m_k steps of value iteration, $V_{k+1} = (T_\gamma^{\pi_k})^{m_k}(V_k)$.

This algorithm guarantees convergence of V_k to V^*.

5.9 Linear Programming Solutions

An alternative approach to value and policy iteration is the linear programming method. Here the optimal control problem is formulated as a linear program (LP), which can be solved efficiently using standard LP solvers. There are two formulations: primal and dual. As this method is less related to learning we will only sketch it briefly.

5.9.1 Some Background on Linear Programming

A Linear Program (LP) is an optimization problem that involves minimizing (or maximizing) a linear objective function subject to linear constraints. A standard form of a LP is

$$\text{minimize } b^T x, \quad \text{subject to } Ax \geq c, x \geq 0. \tag{5.5}$$

where $x = (x_1, x_2, \ldots, x_n)^T$ is a vector of real variables arranged as a column vector. The set of constraints is linear and defines a *convex polytope* in $\mathbb{R}^n$, namely a closed and convex

set U that is the intersection of a finite number of half-spaces. U has a finite number of vertices, which are points that cannot be generated as a convex combination of other points in U. If U is bounded, it equals the convex combination of its vertices. It can be seen that an optimal solution (if finite) will be in one of these vertices.

The LP problem has been extensively studied, and many efficient solvers exist. In 1947, Danzig introduced the Simplex algorithm, which essentially moves greedily along neighboring vertices. In the 1980's effective algorithms (interior point and others) were introduced which had polynomial time guarantees.

Duality: The *dual* of the LP in (5.5) is defined as the following LP:

$$\text{maximize } c^T y, \quad \text{subject to } A^T y \leq b, y \geq 0. \tag{5.6}$$

The two dual LPs have the same optimal value, and the solution of one can be obtained from that of the other. The common optimal value can be understood by the following computation:

$$\begin{aligned}\min_{x\geq 0, Ax\geq c} b^T x &= \min_{x\geq 0}\max_{y\geq 0}\left\{b^T x + y^T(c-Ax)\right\} \\ &= \max_{y\geq 0}\min_{x\geq 0}\left\{c^T y + x^T(b-Ay)\right\} = \max_{y\geq 0, Ay\leq b} c^T x.\end{aligned}$$

where the second equality follows by the min-max theorem.

Note: For an LP of the form:

$$\text{minimize } b^T x, \quad \text{subject to } Ax \geq c,$$

the dual is

$$\text{maximize } c^T y, \quad \text{subject to } A^T y = b, y \geq 0.$$

5.9.2 The Primal LP

Recall that V^*satisfies the optimality equations:

$$V(s) = \max_{a\in A}\left\{r(s,a) + \gamma \sum\nolimits_{s'\in S} p(s'|s,a)V(s')\right\}, \quad s \in S.$$

Proposition 5.3. *V^* is the smallest function (component-wise) that satisfies the following set of inequalities:*

$$v(s) \geq r(s,a) + \gamma \sum\nolimits_{s'\in S} p(s'|s,a)v(s'), \quad \forall s, a. \tag{5.7}$$

Proof. Suppose $v = (v(s))$ satisfies (5.7). That is, $v \geq T^\pi v$ for every stationary policy. Then by the monotonicity of T^π,

$$v \geq T^\pi(v) \;\; \Rightarrow \;\; T^\pi(v) \; \geq (T^\pi)^2(v) \;\; \Rightarrow \;\; \ldots \;\Rightarrow \;\; (T^\pi)^k(v) \; \geq (T^\pi)^{k+1}v,$$

so that

$$v \geq T^\pi(v) \geq (T^\pi)^2(v) \geq \ldots \geq \lim_{n\to\infty} (T^\pi)^n(v) = V^\pi.$$

Now, if we take π as the optimal policy we obtain $v \geq V^*$ (component-wise). □

It follows from Proposition 5.3 that V^* is the solution of the following linear program:

Primal LP:

$$\min_{(v(s))} \; \sum_s v(s), \quad \text{subject to (5.7)} \; .$$

Note that the number of inequality constraints is $N_S \times N_A$.

5.9.3 The Dual LP

The dual of our Primal LP turns out to be:

Dual LP:

$$\max_{(f_{s,a})} \quad \sum_{s,a} f_{s,a} r(s,a)$$

subject to:

$$f_{s,a} \geq 0 \quad \forall s, a$$

$$\sum_{s,a} f_{s,a} = \tfrac{1}{1-\gamma}$$

$$p_0(s') + \gamma \sum_{s,a} p(s'|s,a) f_{s,a} = \sum_a f_{s',a} \quad \forall s' \in S$$

where $p_0 = (p_0(s'))$ is any probability vector (usually taken as a 0/1 vector).

Interpretation:

1. The variables $f_{s,a}$ correspond to the "state action frequencies" (for a given policy):

$$f_{s,a} \sim E(\sum_{t=0}^{\infty} \gamma^t I_{\{s_t=s, a_t=a\}}) = \sum_{t=0}^{\infty} \gamma^t P(s_t = s, a_t = a),$$

and $p_0(s') \sim p(s_0 = s')$ is the initial state distribution.

2. It is easy to see that the discounted return can be written in terms of $f_{s,a}$ as $\sum_s f_{s,a} r(s,a)$, which is to be maximized.

3. The above constraints easily follow from the definition of $f_{s,a}$.

Further comments:

- The optimal policy can by obtained directly from the solution of the dual using:

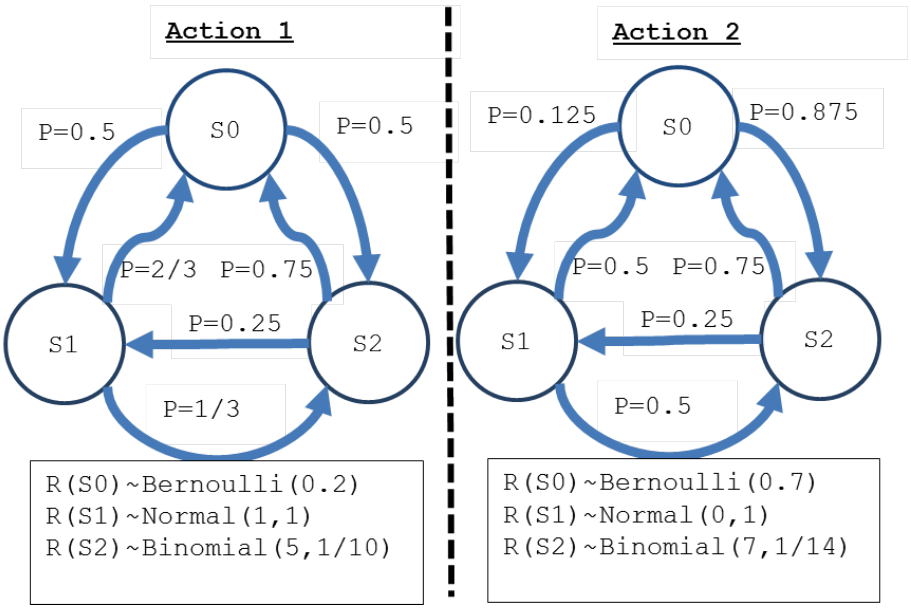
$$\pi(a|s) = \frac{f_{s,a}}{f_s} \equiv \frac{f_{s,a}}{\sum_a f_{s,a}}$$

  This policy can be stochastic if the solution to the LP is not unique. However, it will be deterministic even in that case if we choose f as an *extreme* solution of the LP.

- The number of constraints in the dual is $N_S N_A + (N_S + 1)$. However, the inequality constraints are simpler than in the primal.

5.10 Exercises

Exercise 5.1. You are inside a shady casino with your not so bright friend Jack. You sit at the first table you see and the dealer offers you the following game: he presents you with a Markov Decision Process where you start at s_0 and can take one of two actions in each state. The transition and rewards for each action are given as follows:



1. You allow Jack to play a few rounds. Since 21 is his favorite number, Jack starts with the action 2, followed by the action 1 then again action 2 and so on. What is Jack's expected reward after 3 rounds (i.e., 3 actions)?

2. Jack changes his strategy and starts a new game (at s_0) choosing the action to be either 1 or 2 with equal probability. What will be Jack's expected reward after 3 rounds now? What is the induced stationary policy over the states?

3. Write and solve Bellman equations for 3 rounds. What is the optimal policy?

4. Assuming each round there is a β probability of getting thrown out of the casino, write down the infinite horizon cumulative reward. Conclude the connection between the discount factor and the death rate of a process.

5. Write the Bellman equations for the infinite horizon discounted case in this problem.

Exercise 5.2 (**Modeling an Inventory MDP**)**.** In this question we will model resource allocation problems as MDPs. For each given scenario, write down what are the corresponding states, actions, state-transitions and reward. Also, write down a suitable performance criteria.

Remark: there may be multiple ways to model each scenario. Write down what you think is the most reasonable.

1. Consider managing a hot-dog stand. At each hour, starting from 08:00, you decide how many hot-dogs to order from your supplier, each costing c, and they arrive instantly. At each hour, the number of hot-dog costumers is a random variable with Poisson distribution with rate r, and each customer buys a hot-dog for price p. At time 22:00 you close the stand, and throw away the remaining unsold hot-dogs.

2. Consider scenario (1), but now each supplied hot-dog can only stay fresh for three hours, and then it has to be thrown away.

3. Consider scenario (1), but now during 12:00-14:00 costumers arrive at double rate.

4. Consider scenario (1), but now the stand is operated non-stop 24 hours a day. In addition, there is a yearly inflation ratio of 3%.

5. Consider scenario (4), but now during 12:00-14:00 costumers arrive at double rate.

Exercise 5.3. Prove the following equality (from Section 5.2 of the lecture notes)

$$\begin{aligned} V^{\pi}(s) &\triangleq E^{\pi}(\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) | s_0 = s) \\ &= E^{\pi}(\sum_{t=1}^{\infty} \gamma^{t-1} r(s_t, a_t) | s_1 = s). \end{aligned}$$

Exercise 5.4 (**The $c\mu$ rule**)**.** Assume N jobs are scheduled to run on a single server. At each time step ($t = 0, 1, 2,)$, the sever may choose one of the remaining unfinished jobs to process. If job i is chosen, then with probability $\mu_i > 0$ it will be completed, and removed from the system; otherwise the job stays in the system, and remains in an unfinished state.

Notice that the job service is memoryless - the probability of a job completion is independent of the number of times it has been chosen. Each job is associated with a waiting cost $c_i > 0$ that is paid for each time step that the job is still in the system. The server's goal is minimizing the total cost until all jobs leave the system.

1. Describe the problem as a Markov decision process. Write Bellman's equation for this problem.

2. Show that the optimal policy is choosing at each time step $i^* = \arg\max_i c_i\mu_i$ (from the jobs that are still in the system).

   Hint: Compute the value function for the proposed policy and show that it satisfies the Bellman equation.

Remark: the $c\mu$ law is a fundamental result in queuing theory, and applies also to more general scenarios.

Exercise 5.5 (**Blackjack**)**.** Black Jack is a popular casino card game. The object is to obtain a hand with the maximal sum of card values, but without exceeding 21. All face cards count as 10, and the ace counts as 11 (unlike the original game). In our version, each player competes independently against the dealer, and the card deck is infinite (i.e., the probability of drawing a new card from the deck does not depend on the cards in hand).

The game begins with two cards dealt to the player and one to the dealer. If the player starts with 21 it is called a natural (an ace and a 10 card), and he wins (reward = 1). If the player did not start with 21 he can request additional cards one by one (hits), until he either chooses to stop (sticks) or exceeds 21 (goes bust). If he goes bust, he loses (reward=-1), if he sticks - then it becomes the dealer's turn. The dealer first draws a second card. Then, the dealer hits or sticks according to a fixed policy: he sticks on any sum of 17 or greater. If the dealer busts, then the player wins (reward = 1). Otherwise, the outcome–win, lose, or draw–is determined by whose final sum is closer to 21.

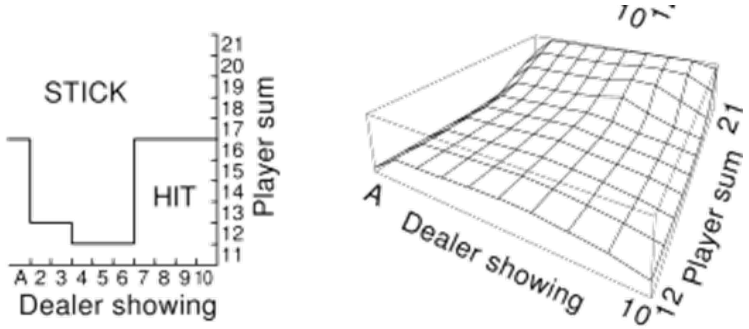
We represent a state as (X, Y) where X is the current player sum and Y is the dealer's first card.

1. Describe the problem as a Markov decision process. What is the size of the state space?

2. Use value iteration to solve the MDP. Plot the optimal value function V^* as a function of (X, Y).

3. Use the optimal value function to derive an optimal policy. Plot the optimal policy as follows: for each value of the dealer's card (Y), plot the minimal value for which the policy sticks.

Here's an example of the plots you should provide (the values should be different though)

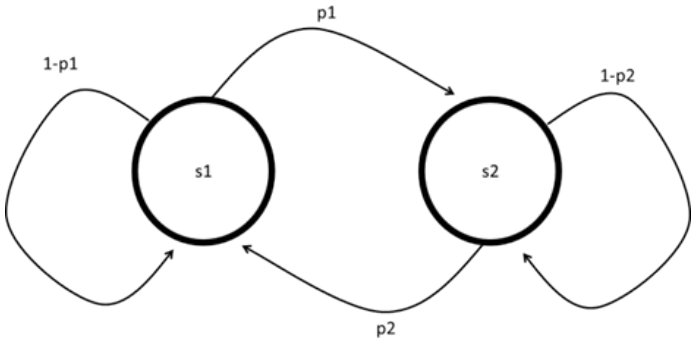


Exercise 5.6 (**DP operator not contracting in Euclidean norm**)**.** Recall the fixed-policy DP operator T^{π} defined as (see Section 5.4.3)

$$\left(T^{\pi}\left(J\right)\right)\left(s\right)=r\left(s,\pi(s)\right)+\gamma\sum_{s'\in S}p\left(s'|s,\pi(s)\right)J\left(s'\right),$$

where $\gamma<1$. We have seen that T^{π} is a contraction in the sup-norm. Show that T^{π} is not necessarily a contraction in the Euclidean norm.

Hint: one possible approach is to consider the following 2-state MDP, and choose appropriate values for p_1, p_2, γ to obtain a contradiction to the contraction property.



Exercise 5.7 (**Contraction of** $(T^{*})^{k}$)**.** Recall that the Bellman operator T^{*} defined by

$$\left(T^{*}\left(J\right)\right)\left(s\right)=\max_{a\in A}\left\{r\left(s,a\right)+\gamma\sum_{s'\in S}p\left(s'|s,a\right)J\left(s'\right)\right\}$$

is a γ-contraction. We will show that $\left(T^{*}\right)^{k}$ is a γ^{k}-contraction.

1. For some J and $\bar{J}$ let $c = \max_s \left|J(s) - \bar{J}(s)\right|$. Show that

$$(T^*)^k (J - ce) \leq (T^*)^k \left(\bar{J}\right) \leq (T^*)^k (J + ce), \tag{5.8}$$

   where e is a vector of ones.

2. Now use (5.8) to show that $(T^*)^k$ is a γ^k-contraction.

Exercise 5.8 (**Second moment and variance of return**)**.** In the lectures we have defined the value function $V^\pi(s)$ as the expected discounted return when starting from state s and following policy π,

$$V^\pi(s) = E^{\pi,s}(\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)).$$

We have seen that $V^\pi(s)$ satisfies a set of $|S|$ linear equations (Bellman equation)

$$V^\pi(s) = r(s, \pi(s)) + \gamma \sum\nolimits_{s' \in S} p(s'|s, \pi(s)) V^\pi(s'), \quad s \in S.$$

We now define $M^\pi(s)$ as the second moment of the discounted return when starting from state s and following policy π,

$$M^\pi(s) = E^{\pi,s}\left(\left(\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right)^2\right).$$

1. We will show that $M^\pi(s)$ satisfies a 'Bellman like' set of equations. Write an expression for $M^\pi(s)$ that has a linear dependence on M^π and V^π.

   Hint: start by following the derivation of the Bellman equation for V^π.

2. How many equations are needed to solve in order to calculate $M^\pi(s)$ for all $s \in S$?

3. We now define $W^\pi(s)$ as the variance of the discounted return when starting from state s and following policy π,

$$W^\pi(s) = \mathrm{Var}^{\pi,s}\left(\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right).$$

   Explain how $W^\pi(s)$ may be calculated.

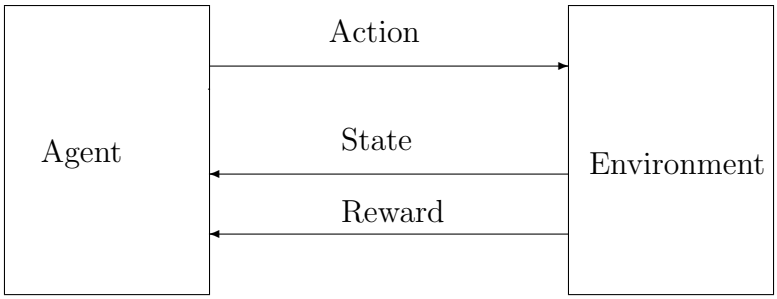
Exercise 5.9. Consider the modified policy iteration scheme of Section 5.8.3. Show that extreme values of m_k (which?) reduce this algorithm to the standard Value Iteration or Policy Iteration.

Chapter 6

Reinforcement Learning – Basic Algorithms

6.1 Introduction

RL methods essentially deal with the solution of (optimal) control problems using on-line measurements. We consider an agent who interacts with a dynamic environment, according to the following diagram:



Our agent usually has only partial knowledge of its environment, and therefore will use some form of *learning* scheme, based on the observed signals. To start with, the agent needs to use some parametric *model* of the environment. We shall use the model of a stationary MDP, with given state space and actions space. However, the state transition matrix $P = (p(s'|s,a))$ and the immediate reward function $r = (r(s,a,s'))$ may not be given. We shall further assume the the observed signal is indeed the state of the dynamic proceed (fully observed MDP), and that the reward signal is the immediate reward r_t, with mean $r(s_t, a_t)$.

It should be realized that this is an *idealized* model of the environment, which is used by the agent for decision making. In reality, the environment may be non-stationary, the actual state may not be fully observed (or not even be well defined), the state and action spaces may be discretized, and the environment may contain other (possibly learning) decision makers who are not stationary. Good learning schemes should be designed with an eye towards robustness to these modelling approximations.

Learning Approaches: The main approaches for learning in this context can be classified as follows:

- Indirect Learning: Estimate an explicit model of the environment ($\hat{P}$ and $\hat{r}$ in our case), and compute an optimal policy for the estimated model ("Certainty Equivalence" and R-MAX that we saw a few lectures ago).
- Direct Learning: The optimal control policy is learned without first learning an explicit model. Such schemes include:

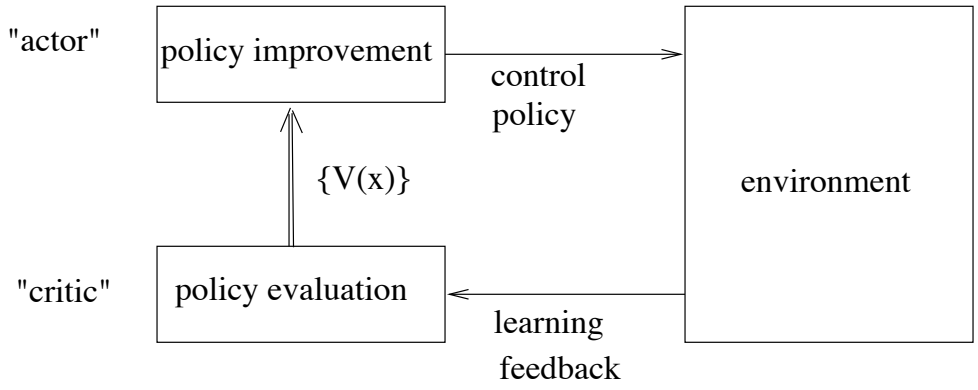
  a. Search in policy space: Genetic Algorithms, Policy Gradient....

  b. Value-function based learning, related to Dynamic Programming principles: Temporal Difference (TD) learning, Q-learning, etc.

RL initially referred to the latter (value-based) methods, although today the name applies more broadly. Our focus in the chapter will be on this class of algorithms.

Within the class of value-function based schemes, we can distinguish two major classes of RL methods.

1. **Policy-Iteration based schemes ("actor-critic" learning):**



   The "policy evaluation" block essentially computes the value function V^π under the current policy (assuming a fixed, stationary policy). Methods for policy evaluation include:

(a) "Monte Carlo" policy evaluation.

(b) Temporal Difference methods - TD(λ), SARSA, etc.

The "actor" block performs some form of policy improvement, based on the policy iteration idea: $\bar{\pi} \in \arg\max\{r + P^{\pi}V^{\pi}\}$. In addition, it is responsible for implementing some "exploration" process.

2. **Value-Iteration based Schemes:** These schemes are based on some on-line version of the value-iteration recursions: $V^*_{t+1} = \max_{\pi}[r^{\pi} + P^{\pi}V^*_t]$. The basic learning algorithm in this class is *Q-learning*.

6.2 Example: Deterministic Q-Learning

To demonstrate some key ideas, we start with a simplified learning algorithm that is suitable for a *deterministic* MDP model, namely:

$$\begin{aligned} s_{t+1} &= f(s_t, a_t) \\ r_t &= r(s_t, a_t) \end{aligned}$$

We consider the discounted return criterion:

$$\begin{aligned} V^{\pi}(s) &= \sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\,, \quad \text{given } s_0 = s, a_t = \pi(s_t) \\ V^*(s) &= \max_{\pi} V^{\pi}(s) \end{aligned}$$

Recall our definition of the Q-function (or *state-action value function*), specialized to the present deterministic setting:

$$Q(s,a) = r(s,a) + \gamma V^*(f(s,a))$$

The optimality equation is then

$$V^*(s) = \max_a Q(s,a)$$

or, in terms of Q only:

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(f(s,a), a')$$

Our learning algorithm runs as follows:

- *Initialize:* Set $\hat{Q}(s,a) = Q_0(s,a)$, for all s, a.

- At each stage $n = 0, 1, \ldots$:
  - Observe s_n, a_n, r_n, s_{n+1}.
  - Update $\hat{Q}(s_n, a_n)$: $\hat{Q}(s_n, a_n) := r_n + \gamma \max_{a'} \hat{Q}(s_{n+1}, a')$

We note that this algorithm does not tell us how to choose the actions a_n. The following result is from [Mitchell, Theorem 3.1].

Theorem 6.1 (Convergence of Q-learning for deterministic MDPS)**.**
Assume a deterministic MDP model. Let $\hat{Q}_n(s, a)$ denote the estimated Q-function before the n-th update. If each state-action pair is visited infinitely-often, then $\lim_{n\to\infty} \hat{Q}_n(s, a) = Q(s, a)$, for all (s, a).

Proof. Let

$$\Delta_n \triangleq \|\hat{Q}_n - Q\|_\infty = \max_{s,a} |\hat{Q}_n(s, a) - Q(s, a)| \,.$$

Then at every stage n:

$$\begin{aligned} |\hat{Q}_{n+1}(s_n, a_n) - Q(s_n, a_n)| &= |r_n + \gamma \max_{a'} \hat{Q}_n(s_{n+1}, a') - (r_n + \gamma \max_{a''} Q(s_{n+1}, a''))| \\ &= \gamma |\max_{a'} \hat{Q}_n(s_{n+1}, a') - \max_{a''} Q(s_{n+1}, a'')| \\ &\le \gamma \max_{a'} |\hat{Q}_n(s_{n+1}, a') - Q(s_{n+1}, a')| \;\le\; \gamma \Delta_n \,. \end{aligned}$$

Consider now some interval $[n_1, n_2]$ over which all state-action pairs (s, a) appear at least once. Using the above relation and simple induction, it follows that $\Delta_{n_2} \le \gamma \Delta_{n_1}$. Since $\gamma < 1$ and since there is an infinite number of such intervals by assumption, it follows that $\Delta_n \to 0$. □

Remarks:

1. The algorithm allows the use of an arbitrary policy to be used during learning. Such as algorithm is called *Off Policy*. In contrast, *On-Policy* algorithms learn the properties of the policy that is actually being applied.

2. We further note that the "next-state" $s' = s_{n+1}$ of stage n need not coincide with the current state s_{n+1} of stage $n + 1$. Thus, we may skip some sample, or even choose s_n at will at each stage. This is a common feature of off-policy schemes.

3. A basic requirement in this algorithm is that all state-action pairs will be samples "often enough". To ensure that we often use a specific *exploration* algorithm or method. In fact, the speed of convergence may depend critically on the efficiency of exploration. We shall discuss this topic in detail further on.

6.3 Policy Evaluation: Monte-Carlo Methods

Policy evaluation algorithms are intended to estimate the value functions V^π or Q^π for a given policy π. Typically these are on-policy algorithms, and the considered policy is assumed to be stationary (or "almost" stationary). Policy evaluation is typically used as the "critic" block of an actor-critic architecture.

Direct Monte-Carlo methods are the most straight-forward, and are considered here mainly for comparison with the more elaborate ones. Monte-Carlo methods are based on the simple idea of averaging a number of random samples of a random quantity in order to estimate its average.

Let π be a fixed stationary policy. Assume we wish to evaluate the value function V^π, which is either the discounted return:

$$V^\pi(s) = E^\pi(\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)|s_0 = s)$$

or the total return for an SSP (or *episodial*) problem:

$$V^\pi(s) = E^\pi(\sum_{t=0}^{T} r(s_t, a_t)|s_0 = s)$$

where T is the (stochastic) termination time, or time of arrival to the terminal state.

Consider first the episodial problem. Assume that we operate (or simulate) the system with the policy π, for which we want to evaluate V^π. Multiple trials may be performed, starting from arbitrary initial conditions, and terminating at T (or truncated before).

After visiting state s, say at time t_s, we add-up the total cost until the target is reached:

$$\hat{v}(s) = \sum_{t=t_s}^{T} R_t \,.$$

After k visits to s, we have a sequence of total-cost estimates:

$$\hat{v}_1(s), \cdots, \hat{v}_k(s) \,.$$

We can now compute our estimate:

$$\hat{V}_k(s) = \frac{1}{k} \sum_{i=1}^{k} \hat{v}_i(s) \,.$$

By repeating these procedure for all states, we estimate $V^\pi(\cdot)$.

State counting options: Since we perform multiple trials and each state can be visited several times per trial, there are several options regarding the visits that will be counted:

a. Compute $\hat{V}(s)$ only for initial states ($s_0 = s$).

b. Compute $\hat{V}(s)$ each time s is visited.

c. Compute $\hat{V}(s)$ only on first visit of s at each trial.

Method (b) gives the largest number of samples, but these may be correlated (hence, lead to non-zero bias for finite times). But in any case, $\hat{V}_k(s) \to V^\pi(s)$ is guaranteed as $k \to \infty$. Obviously, we still need to guarantee that each state is visited enough – this depends on the policy π and our choice of initial conditions for the different trials.

Remarks:

1. The explicit averaging of the $\hat{v}_k$'s may be replaced by the iterative computation:

$$\hat{V}_k(s) = \hat{V}_{k-1}(s) + \alpha_k \Big[\hat{v}_k(s) - \hat{V}_{k-1}(s) \Big] ,$$

   with $\alpha_k = \frac{1}{k}$. Other choices for α_k are also common, e.g. $\alpha_k = \frac{\gamma}{k}$, and $\alpha_k = \epsilon$ (non-decaying gain, suitable for non-stationary conditions).

2. For discounted returns, the computation needs to be truncated at some finite time T_s, which can be chosen large enough to guarantee a small error:

$$\hat{v}(s) = \sum_{t=t_s}^{T_s} (\gamma)^{t-t_s} R_t \, .$$

6.4 Policy Evaluation: Temporal Difference Methods

6.4.1 The TD(0) Algorithm

Consider the total-return (SSP) problem with $\gamma = 1$. Recall the fixed-policy Value Iteration procedure of Dynamic Programming:

$$V_{n+1}(s) = E^\pi (r(s,a) + V_n(s')) = r(s, \pi(s)) + \sum_{s'} p(s'|s, \pi(s)) V_n(s') \, , \quad \forall s \in S,$$

or $V_{n+1} = r^\pi + P^\pi V_n$, which converges to V^π.

Assume now that r^π and P^π are not given. We wish to devise a "learning version" of the above policy iteration.

Let us run or simulate the system with policy π. Suppose we start with some estimate $\hat{V}$ of V^π. At time n, we observe s_n, r_n and s_{n+1}. We note that $[r_n + \hat{V}(s_{n+1})]$ is an unbiased estimate for the right-hand side of the value iteration equation, in the sense that

$$E^\pi(r_n + \hat{V}(s_{n+1})|s_n) = r(s_n, \pi(s_n)) + \sum_{s'} p(s'|s_n, \pi(s_n))V_n(s')$$

However, this is a *noisy* estimate, due to randomness in r and s'. We therefore use it to modify $\hat{V}$ only slightly, according to:

$$\begin{aligned}\hat{V}(s_n) &:= (1-\alpha_n)\hat{V}(s_n) + \alpha_n[r_n + \hat{V}(s_{n+1})] \\ &= \hat{V}(s_n) + \alpha_n[r_n + \hat{V}(s_{n+1}) - \hat{V}(s_n)]\end{aligned}$$

Here α_n is the *gain* of the algorithm. If we define now

$$d_n \triangleq r_n + \hat{V}(s_{n+1}) - \hat{V}(s_n)$$

we obtain the update rule:

$$\hat{V}(s_n) := \hat{V}(s_n) + \alpha_n d_n$$

d_n is called the *Temporal Difference*. The last equation defines the TD(0) algorithm.

Note that $\hat{V}(s_n)$ is updated on basis of $\hat{V}(s_{n+1})$, which is itself an estimate. Thus, TD is a "bootstrap" method: convergence of $\hat{V}$ at each states s is inter-dependent with other states.

Convergence results for TD(0) (preview):

1. If $\alpha_n \searrow 0$ at suitable rate ($\alpha_n \approx 1/$no. of visits to s_n), and each state is visited i.o., then $\hat{V}_n \to V^\pi$ w.p. 1.
2. If $\alpha_n = \alpha_0$ (a small positive constant) and each state is visited i.o., then $\hat{V}_n$ will "eventually" be close to V^π with high probability. That is, for every $\epsilon > 0$ and $\delta > 0$ there exists α_0 small enough so that

$$\lim_{n\to\infty} \text{Prob}(|\hat{V}_n - V^\pi| > \epsilon) \leq \delta\,.$$

6.4.2 TD with ℓ-step look-ahead

TD(0) looks only one step in the "future" to update $\hat{V}(s_n)$, based on r_n and $\hat{V}(s_{n+1})$. Subsequent changes will not affect $\hat{V}(s_n)$ until s_n is visited again.

Instead, we may look ℓ steps in the future, and replace d_n by

$$\begin{aligned}d_n^{(\ell)} &\triangleq \sum_{m=0}^{\ell-1} r_{n+m} + \hat{V}(s_{n+\ell}) - \hat{V}(s_n) \\ &= \sum_{m=0}^{\ell-1} d_{n+m}\end{aligned}$$

where d_n is the one-step temporal difference as before. The iteration now becomes

$$\hat{V}(s_n) := \hat{V}(s_n) + \alpha_n d_n^{(\ell)}.$$

This is a "middle-ground" between TD(0) and Monte-Carlo evaluation!

6.4.3 The TD(λ) Algorithm

Another way to look further ahead is to consider all future Temporal Differences with a "fading memory" weighting:

$$\hat{V}(s_n) := \hat{V}(s_n) + \alpha(\sum_{m=0}^{\infty} \lambda^m d_{n+m}) \tag{6.1}$$

where $0 \leq \lambda \leq 1$. For $\lambda = 0$ we get TD(0); for $\lambda = 1$ we obtain the Monte-Carlo sample!

Note that each run is terminated when the terminal state is reached, say at step T. We thus set $d_n \equiv 0$ for $n \geq T$.

The convergence properties of TD(λ) are similar to TD(0). However, TD(λ) often converges faster than TD(0) or direct Monte-Carlo methods, provided that λ is properly chosen. This has been experimentally observed, especially when function approximation is used for the value function.

Implementations of TD(λ): There are several ways to implement the relation in (6.1).

1. Off-line implementation: $\hat{V}$ is updated using (6.1) at the end of each simulation run, based on the stored (s_t, d_t) sequence from that run.

2. Each d_n is used as soon as becomes available, via the following backward update (also called "on-line implementation"):

$$\hat{V}(s_{n-m}) := \hat{V}(s_{n-m}) + \alpha \cdot \lambda^m d_n\,, \qquad m = 0, \ldots, n\,. \tag{6.2}$$

   This requires only keeping track of the state sequence $(s_t, t \geq 0)$. Note that if some state s appears twice in that sequence, it is updated twice.

3. Eligibility-trace implementation:

$$\hat{V}(s) := \hat{V}(s) + \alpha d_n e_n(s)\,, \qquad s \in S \tag{6.3}$$

   where

$$e_n(s) = \sum_{k=0}^{n} \lambda^{n-k} 1\{s_k = s\}$$

   is called the *eligibility trace* for state s.

The eligibility trace variables $e_n(s)$ can also be computed recursively. Thus, set $e_0(s) = 0 \quad \forall s$, and

$$e_n(s) := \lambda e_{n-1}(s) + 1\{s_n = s\} = \begin{cases} \lambda \cdot e_{n-1}(s) + 1 & \text{if} \quad s = s_n \\ \lambda \cdot e_{n-1}(s) & \text{if} \quad s \neq s_n \end{cases} \tag{6.4}$$

Equations (6.3) and (6.4) provide a fully recursive implementation of TD(λ).

6.4.4 TD Algorithms for the Discounted Return Problem

For γ-discounted returns, we obtain the following equations for the different TD algorithms:

1. TD(0):

$$\begin{aligned} \hat{V}(s_n) &:= (1-\alpha)\hat{V}(s_n) + \alpha[r_n + \gamma \hat{V}(s_{n+1}] \\ &= \hat{V}(s_n) + \alpha \cdot d_n, \end{aligned}$$

   with $d_n \triangleq r_n + \gamma V(s_{n+1}) - V(s_n)$.

2. ℓ-step look-ahead:

$$\begin{aligned} \hat{V}(s_n) &:= (1-\alpha)\hat{V}(s_n) + \alpha[r_n + \gamma r_{n+1} + \cdots + \gamma^{\ell} V_{n+\ell}] \\ &= \hat{V}(s_n) + \alpha[d_n + \gamma d_{n+1} + \cdots + \gamma^{\ell-1} d_{n+\ell-1}] \end{aligned}$$

3. TD(λ):

$$\hat{V}(s_n) := \hat{V}(s_n) + \alpha \sum_{k=0}^{\infty} (\gamma\lambda)^k d_{n+k} \,.$$

   The eligibility-trace implementation is:

$$\begin{aligned} \hat{V}(s) &:= \hat{V}(s) + \alpha d_n e_n(s) \,, \\ e_n(s) &:= \gamma\lambda e_{n-1}(s) + 1\{s_n = s\} \,. \end{aligned}$$

6.4.5 Q-functions and their Evaluation

For policy improvement, what we require is actually the Q-function $Q^\pi(s,a)$, rather than $V^\pi(s)$. Indeed, recall the policy-improvement step of policy iteration, which defines the improved policy $\hat{\pi}$ via:

$$\hat{\pi}(s) \in \arg\max_a \{r(s,a) + \gamma \sum_{s'} p(s'|s,a) V^\pi(s')\} \equiv \arg\max_a Q^\pi(s,a) \,.$$

How can we estimate Q^π?

1. Using $\hat{V}^\pi$: If we know the one-step model parameters r and p, we may estimate $\hat{V}^\pi$ as above and compute

$$\hat{Q}^\pi(s,a) \stackrel{\triangle}{=} r(s,a) + \gamma \sum p(s'|s,a)\hat{V}^\pi(s') \,.$$

When the model is not known, this requires to estimate r and p on-line.

2. Direct estimation of Q^π: This can be done using the same methods as outlined for $\hat{V}^\pi$, namely Monte-Carlo or TD methods. We mention the following:

The SARSA algorithm: This is the equivalent of TD(0). At each stage we observe $(s_n, a_n, r_n, s_{n+1}, a_{n+1})$, and update

$$\begin{aligned} Q(s_n,a_n) &:= Q(s_n,a_n) + \alpha_n \cdot d_n \\ d_n &= r_n + \gamma Q(s_{n+1},a_{n+1}) - Q(s_n,a_n), \text{where } a_{n+1} = \pi(s_{n+1}). \end{aligned}$$

Similarly, the SARSA(λ) algorithm uses

$$\begin{aligned} Q(s,a) &:= Q(s,a) + \alpha_n(s,a) \cdot d_n e_n(s,a) \\ e_n(s,a) &:= \gamma\lambda e_{n-1}(s,a) + 1\{s_n = 1, a_n = a\} \,. \end{aligned}$$

Note that:
– The estimated policy π must be the one used ("on-policy" scheme).
– More variables are estimated in Q than in V.

6.5 Policy Improvement

Having studied the "policy evaluation" block of the actor/critic scheme, we turn to the policy improvement part.

Ideally, we wish to implement policy iteration through learning:

(i) Using policy π, evaluate $\hat{Q} \approx Q^\pi$. Wait for convergence.

(ii) Compute $\hat{\pi} = \arg\max \hat{Q}$ (the "greedy policy" w.r.t. $\hat{Q}$).

Problems:

a. Convergence in (i) takes infinite time.

b. Evaluation of $\hat{Q}$ requires trying all actions – typically requires an exploration scheme which is richer than the current policy π.

To solve (a), we may simply settle for a finite-time estimate of Q^π, and modify π every (sufficiently long) finite time interval. A more "smooth" option is to modify π slowly in the "direction" of the maximizing action. Common options include:

(i) Gradual maximization: If a^* maximizes $\hat{Q}(s,a)$, where s is the state currently examined, then set

$$\begin{cases} \pi(a^*|s) := \pi(a^*|s) + \alpha \cdot [1 - \pi(a^*|s)] \\ \pi(a|s) := \pi(a|s) - \alpha \cdot \pi(a|s), \quad a \neq a^* \,. \end{cases}$$

Note that π is a *randomized* stationary policy, and indeed the above rule keeps $\pi(\cdot|s)$ as a probability vector.

(ii) Increase probability of actions with high Q: Set

$$\pi(a|s) = \frac{e^{\beta(s,a)}}{\sum_{a'} e^{\beta(s,a')}}$$

(a Boltzmann-type distribution), where β is updated as follows:

$$\beta(s,a) := \beta(s,a) + \alpha[\hat{Q}(s,a) - \hat{Q}(s,a_0)].$$

Here a_0 is some arbitrary (but fixed) action.

(iii) "Pure" actor-critic: Same Boltzmann-type distribution is used, but now with

$$\beta(s,a) := \beta(s,a) + \alpha[r(s,a) + \gamma\hat{V}(s') - \hat{V}(s)]$$

for $(s,a,s') = (s_n, a_n, s_{n+1})$. Note that this scheme uses directly $\hat{V}$ rather than $\hat{Q}$. However it is more noisy and harder to analyze than other options.

To address problem (b) (exploration), the simplest approach is to superimpose some randomness over the policy in use. Simple local methods include:

(i) ϵ-exploration: Use the nominal action a_n (e.g., $a_n = \arg\max_a Q(s_n, a)$) with probability $(1-\epsilon)$, and otherwise (with probability ϵ) choose another action at random. The value of ϵ can be reduced over time, thus shifting the emphasis from exploration to exploitation.

(ii) Softmax: Actions at state s are chosen according to the probabilities

$$\pi(a|s) = \frac{e^{Q(s,a)/\theta}}{\sum_a e^{Q(s,a)/\theta}} \,.$$

θ is the "temperature" parameter, which may be reduced gradually.

(iii) The above "gradual maximization" methods for policy improvement.

These methods however may give slow convergence results, due to their local (state-by-state) nature.

Another simple (and often effective) method for exploration relies on the principle of *optimism in the face of uncertainty*. For example, by initializing $\hat{Q}$ to high (optimistic) values, we encourage greedy action selection to visit unexplored states.

Convergence analysis for actor-critic schemes is relatively hard. Existing results rely on a *two time scale* approach, where the rate of policy update is assumed much slower than the rate of value-function update.

6.6 Q-learning

Q-learning is the most notable representative of *value iteration* based methods. Here the goal is to compute directly the *optimal* value function. These schemes are typically *off-policy* methods – learning the optimal value function can take place under any policy (subject to exploration requirements).

Recall the definition of the (optimal) Q-function:

$$Q(s,a) \triangleq r(s,a) + \gamma \sum_{s'} p(s'|s,a) V^*(s') \,.$$

The optimality equation is then $V^*(s) = \max_a Q(s,a)\,,\ s \in S$, or in terms of Q only:

$$Q(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) \max_{a'} Q(s',a') \,, \quad s \in S, a \in A \,.$$

The value iteration algorithm is given by:

$$V_{n+1}(s) = \max_a \{ r(s,a) + \gamma \sum_{s'} p(s'|s,a) V_n(s') \} \,, \quad s \in S$$

with $V_n \to V^*$. This can be reformulated as

$$Q_{n+1}(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) \max_{a'} Q_n(s',a') \,, \tag{6.5}$$

with $Q_n \to Q$.

We can now define the on-line (learning) version of the Q-value iteration equation.

The Q-learning algorithm: – initialize $\hat{Q}$.

– At stage n: Observe (s_n, a_n, r_n, s_{n+1}), and let

$$\begin{aligned} \hat{Q}(s_n,a_n) &:= (1-\alpha_n)\hat{Q}(s_n,a_n) + \alpha_n [r_n + \gamma \max_{a'} \hat{Q}(s_{n+1},a')] \\ &= \hat{Q}(s_n,a_n) + \alpha_n [r_n + \gamma \max_{a'} \hat{Q}(s_{n+1},a') - \hat{Q}(s_n,a_n)] \,. \end{aligned}$$

The algorithm is obviously very similar to the basic TD schemes for policy evaluation, except for the maximization operation.

Convergence: If all (s, a) pairs are visited i.o., and $\alpha_n \searrow 0$ at appropriate rate, then $\hat{Q}_n \to Q^*$.

Policy Selection:

- Since learning the Q^* does not depend on optimality of the policy used, we can focus on exploration during learning. However, if learning takes place while the system is in actual operation, we may still need to use a close-to-optimal policy, while using the standard exploration techniques (ϵ-greedy, softmax, etc.).
- When learning stops, we may choose a greedy policy:

$$\hat{\pi}(s) = \max_a \hat{Q}(s, a) \,.$$

Performance: Q-learning is very convenient to understand and implement; however, convergence may be slower than actor-critic (TD(λ)) methods, especially if in the latter we only need to evaluate V and not Q.

6.7 Exercises

Exercise 6.1 (**The $c\mu$ rule revisited**). Consider again the job processing domain of Exercise 5.4:

N jobs are scheduled to run on a single server. At each time step ($t = 0, 1, 2,)$, the sever may choose one of the remaining unfinished jobs to process. If job i is chosen, then with probability $\mu_i > 0$ it will be completed, and removed from the system; otherwise the job stays in the system, and remains in an unfinished state. Notice that the job service is memoryless - the probability of a job completion is independent of the number of times it has been chosen. Each job is associated with a waiting cost $c_i > 0$ that is paid for each time step that the job is still in the system. The server's goal is minimizing the total cost until all jobs leave the system.

This time, we will solve the problem numerically, testing the various dynamic programming and reinforcement learning algorithms we have learned so far.

We consider the following specific case, with $N = 5$:

i	**1**	**2**	**3**	**4**	**5**
μ_i	0.6	0.5	0.3	0.7	0.1
c_i	1	4	6	2	9

Part 1 - Planning In this part all the Matlab functions may use the true model (i.e., the μ_i's and c_i's).

1. How many states and actions are in the problem?

Let S denote the state space. A deterministic policy π is a mapping from states to actions. In our case, it will be represented in Matlab as a vector of length $|S|$, in which each element denotes the selected action $(1, \ldots, N)$ for the corresponding state.

2. Write a function (in Matlab) that take as input a policy, and returns the corresponding value function V^π, also represented as a vector of length $|S|$. You can solve it either directly by matrix inversion, or iteratively. Remember that the value of the terminal state (no more jobs left) is zero by definition.

3. Plot the values of the policy π_c that selects the job with the maximal cost c_i, from the remaining unfinished jobs.

4. Write a function that calculates the optimal policy π^* using the policy iteration algorithm, and execute it, starting from the initial policy π_c. For each iteration of the algorithm, plot the value of the initial state s_0 (no jobs completed) for the current policy. How many steps are required for convergence?

5. Compare the optimal policy π^* obtained using policy iteration to the $c\mu$ law. Also plot V^{π^*} vs. V^{π_c}.

6. Write a simulator of the problem: a function that takes in a state s and action a, and returns the cost of the state $c(s)$, and a random next state s', distributed according to the transition probabilities of the problem.

Part 2 - Learning In this part the learning algorithms cannot use the true model parameters, but only have access to the simulator function written above.

7. **Policy evaluation:** consider again the policy π_c, and use the TD(0) algorithm to learn the value function V^{π_c}. Start from $\hat{V}_{TD}(s) = 0$ for all states. Experiment with several step size α_n schedules:

   (a) $a_n = 1/\left(no.\,of\,visits\,to\,s_n\right)$
   (b) $a_n = 0.01$
   (c) $a_n = \frac{10}{100+(no.\,of\,visits\,to\,s_n)}$

   For each step-size schedule, plot the errors $\left\|V^{\pi_c} - \hat{V}_{TD}\right\|_\infty$ and $\left|V^{\pi_c}(s_0) - \hat{V}_{TD}(s_0)\right|$ as a function of iteration n. Explain the motivation behind each step-size schedule, and how it reflects in the results.

8. Now, run policy evaluation using TD(λ). Choose your favorite step-size schedule, and plot the errors $\left\|V^{\pi_c} - \hat{V}_{TD}\right\|_\infty$ and $\left|V^{\pi_c}(s_0) - \hat{V}_{TD}(s_0)\right|$ as a function of iteration n for several choices of λ. Repeat each experiment 20 times and display average results.

9. **Q-learning:** run Q-learning to find the optimal policy. Start from $\hat{Q}(s,a) = 0$ for all states and actions. Experiment with the 3 previous step-size schedules. Use ϵ-greedy exploration, with $\epsilon = 0.1$.

   For some $\hat{Q}$, let $\pi_{\hat{Q}}$ denote the greedy policy w.r.t. $\hat{Q}$, i.e., $\pi_{\hat{Q}}(s) = \arg\min_a \hat{Q}(s,a)$.

   For each step-size schedule, plot the errors $\|V^* - V^{\pi_{\hat{Q}}}\|_\infty$ and $\left|V^*(s_0) - \arg\min_a \hat{Q}(s_0,a)\right|$ as a function of iteration n. You may use the policy evaluation function you wrote in (2) to calculate $V^{\pi_{\hat{Q}}}$. If this step takes too long, you may plot the errors only for $n = 100, 200, 300, ...$ etc.

10. Repeat the previous Q-learning experiment for your favorite step-size schedule but now with $\epsilon = 0.01$. Compare.

Chapter 7

The Stochastic Approximation Algorithm

7.1 Stochastic Processes – Some Basic Concepts

7.1.1 Random Variables and Random Sequences

Let $(\Omega, \mathcal{F}, P)$ be a probability space, namely:

- Ω is the sample space.
- $\mathcal{F}$ is the event space. Its elements are subsets of Ω, and it is required to be a σ-algebra (includes $\emptyset$ and Ω; includes all countable union of its members; includes all complements of its members).
- P is the probability measure (assigns a probability in [0,1] to each element of $\mathcal{F}$, with the usual properties: $P(\Omega) = 1$, countably additive).

A random variable (RV) X on $(\Omega, \mathcal{F})$ is a function $X : \Omega \to \mathbb{R}$, with values $X(\omega)$. It is required to be *measurable* on $\mathcal{F}$, namely, all sets of the form $\{\omega : X(\omega) \leq a\}$ are events in $\mathcal{F}$.

A vector-valued RV is a vector of RVs. Equivalently, it is a function $X : \Omega \to \mathbb{R}^d$, with similar measurability requirement.

A *random sequence*, or a discrete-time *stochastic process*, is a sequence $(X_n)_{n\geq 0}$ of $\mathbb{R}^d$-valued RVs, which are all defined on the same probability space.

7.1.2 Convergence of Random Variables

A random sequence may converge to a random variable, say to X. There are several useful notions of convergence:

1. Almost sure convergence (or: convergence with probability 1):

$$X_n \to a.s.X \quad \text{if} \quad P\{\lim_{n\to\infty} X_n = X\} = 1\,.$$

2. Convergence in probability:

$$X_n \to pX \quad \text{if} \quad \lim_{n\to\infty} P(|X_n - X| > \epsilon) = 0\,, \forall \epsilon > 0\,.$$

3. Mean-squares convergence (convergence in L^2):

$$X_n \to L^2 X \quad \text{if} \quad E|X_n - X_\infty|^2 \to 0\,.$$

4. Convergence in Distribution:

$$X_n \to DistX \text{ (or } X_n \Rightarrow X) \quad \text{if} \quad Ef(X_n) \to Ef(X)$$

   for every bounded and continuous function f.

The following relations hold:

a. Basic implications: (a.s. or L^2) $\Longrightarrow p \Longrightarrow$ Dist

b. Almost sure convergence is equivalent to

$$\lim_{n\to\infty} P\{\sup_{k\geq n} |X_k - X| > \epsilon) = 0\,, \quad \forall \epsilon > 0\,.$$

c. A useful *sufficient* condition for a.s. convergence:

$$\sum_{n=0}^{\infty} P(|X_n - X| > \epsilon) < \infty\,.$$

7.1.3 Sigma-algebras and information

Sigma algebras (or σ-algebras) are part of the mathematical structure of probability theory. They also have a convenient interpretation as "information sets", which we shall find useful.

- Define $\mathcal{F}_X \triangleq \sigma\{X\}$, the σ-algebra generated by the RV X. This is the smallest σ-algebra that contains all sets of the form $\{X \leq a\} \equiv \{\omega \in \Omega : X(\omega) \leq a\}$.
- We can interpret $\sigma\{X\}$ as carrying all the information in X. Accordingly, we identify

$$E(Z|X) \equiv E(Z|\mathcal{F}_X)\,.$$

  Also, "Z is measurable on $\sigma\{X\}$" is equivalent to: $Z = f(X)$ (with the additional technical requirement that f is a Borel measurable function).

- We can similarly define $\mathcal{F}_n = \sigma\{X_1, \ldots, X_n\}$, etc. Thus,

$$E(Z|X_1, \ldots, X_n) \equiv E(Z|\mathcal{F}_n)\,.$$

- Note that $\mathcal{F}_{n+1} \supset \mathcal{F}_n$: more RVs carry more information, leading $\mathcal{F}_{n+1}$ to be finer, or "more detailed"

7.1.4 Martingales

A *martingale* is defined as follows.

Definition 7.1 (**Martingale**). *A sequence* $(X_k, \mathcal{F}_k)_{k\geq 0}$ *on a given probability space* $(\Omega, \mathcal{F}, P)$ *is a martingale if*

a. $(\mathcal{F}_k)$ *is a "filtration" – an increasing sequence of* σ*-algebras in* $\mathcal{F}$*.*

b. Each RV X_k *is* $\mathcal{F}_k$*-measurable.*

c. $E(X_{k+1}|\mathcal{F}_k) = X_k$ *(P–a.s.).*

Note:

- (a) Property is roughly equivalent to:
  $\mathcal{F}_k$ represents (the information in) some RVs $(Y_0, \ldots, Y_k)$,
  and (b) then means: X_k is a function of $(Y_0, \ldots, Y_k)$.
- A particular case is $\mathcal{F}_n = \sigma\{X_1, \ldots, X_n\}$ (a self-martingale).
- The central property is (c), which says that the conditional mean of X_{k+1} equals X_k. This is obviously stronger than $E(X_{k+1}) = E(X_k)$.
- The definition sometimes requires also that $E|X_n| < \infty$, we shall assume that below.
- Replacing (c) by $E(X_{k+1}|\mathcal{F}_k) \geq X_k$ gives a *submartingale*, while $E(X_{k+1}|\mathcal{F}_k) \leq X_k$ corresponds to a *supermartingale*.

Examples:

a. The simplest example of a martingale is

$$X_k = \sum_{\ell=0}^{k} \xi_\ell\,,$$

with $\{\xi_k\}$ a sequence of 0-mean independent RVs, and $\mathcal{F}_k = \sigma(\xi_0, \ldots, \xi_k)$.

b. $X_k = E(X|\mathcal{F}_k)$, where (F_k) is a given filtration and X a fixed RV.

Martingales play an important role in the convergence analysis of stochastic processes. We quote a few basic theorems (see, for example: A.N. Shiryaev, *Probability,* Springer, 1996).

Theorem 7.1 (**Martingale Inequalities**). *Let$(X_k, \mathcal{F}_k)_{k\geq 0}$ be a martingale. Then for every $\lambda > 0$ and $p \geq 1$*

$$P\left\{\max_{k\leq n} |X_k| \geq \lambda\right\} \leq \frac{E|X_n|^p}{\lambda^p},$$

and for $p > 1$

$$E[(\max_{k\leq n} |X_k|)^p] \leq (\tfrac{p}{p-1})^p E(|X_n|^p)\,.$$

Martingale Convergence Theorems

Theorem 7.2 (**Convergence with Bounded-moments**). *Consider a martingale $(X_k, \mathcal{F}_k)_{k\geq 0}$. Assume that:*

$$E|X_k|^q \leq C \text{ for some } C < \infty,\ q \geq 1 \text{ and all } k.$$

Then $\{X_k\}$ converges (a.s.) to a RV X_∞ (which is finite w.p. 1).

Theorem 7.3 (**Positive Martingale Convergence**). *If $(X_k, \mathcal{F}_k)$ is a positive martingale (namely $X_n \geq 0$), then X_k converges (a.s.) to some RV X_∞.*

Definition 7.2 (**Martingale Difference**). *The sequence $(\xi_k, \mathcal{F}_k)$ is a* martingale difference *sequence if property (c) is replaced by $E(\xi_{k+1}|\mathcal{F}_k) = 0$.*

In this case we have:

Theorem 7.4 (**Martingale Difference Convergence**). *Suppose that for some $0 < q \leq 2$, $\sum_{k=1}^{\infty} \frac{1}{k^q} E(|\xi_k|^q|F_{k-1}) < \infty$ (a.s.). Then $\lim_{n\to\infty} \frac{1}{n}\sum_{k=1}^{n} \xi_k = 0$ (a.s.).*

For example, the conclusion holds if the sequence (ξ_k) is bounded, namely $|\xi_k| \leq C$ for some $C > 0$ (independent of k).

Note:

- It is trivially seen that $(\xi_n \triangleq X_n - X_{n-1})$ is a martingale difference if (X_n) is a martingale.
- More generally, for any sequence (Y_k) and filtration $(\mathcal{F}_k)$, where Y_k is measurable on $\mathcal{F}_k$, the following is a martingale difference:

$$\xi_k \triangleq Y_k - E(Y_k|\mathcal{F}_{k-1})\,.$$

The conditions of the last theorem hold for this ξ_k if either:
(i) $|Y_k| \leq M \ \forall k$ for some constant $M < \infty$,
(ii) or, more generally, $E(|Y_k|^q|\mathcal{F}_{k-1}) \leq M$ (a.s.) for some $q > 1$ and a finite RV M.
In that case we have

$$\frac{1}{n}\sum_{k=1}^{n}\xi_k \equiv \frac{1}{n}\sum_{k=1}^{n}(Y_k - E(Y_k|\mathcal{F}_{k-1})) \to 0 \quad \text{(a.s.)}$$

7.2 The Basic SA Algorithm

The stochastic approximations (SA) algorithm essentially solves a system of (nonlinear) equations of the form

$$h(\theta) = 0,$$

based on noisy measurements of $h(\theta)$.

More specifically, we consider a (continuous) function $h : \mathbb{R}^d \to \mathbb{R}^d$, with $d \geq 1$, which depends on a set of parameters $\theta \in \mathbb{R}^d$. Suppose that h is unknown. However, for each θ we can measure $Y = h(\theta) + \omega$, where ω is some 0-mean noise. The classical SA algorithm (Robbins-Monro, 1951) is of the form

$$\begin{aligned}\theta_{n+1} &= \theta_n + \alpha_n Y_n \\ &= \theta_n + \alpha_n[h(\theta_n) + \omega_n], \quad n \geq 0\,.\end{aligned}$$

Here α_n is the algorithm the step-size, or *gain*.

Obviously, with zero noise ($\omega_n \equiv 0$) the stationary points of the algorithm coincide with the solutions of $h(\theta) = 0$. Under appropriate conditions (on α_n, h and ω_n) the algorithm indeed can be shown to converge to a solution of $h(\theta) = 0$.

References:

1. H. Kushner and G. Yin, *Stochastic Approximation Algorithms and Applications*, Springer, 1997.

2. V. Borkar, *Stochastic Approximation: A Dynamic System Viewpoint*, Hindustan, 2008.

3. J. Spall, *Introduction to Stochastic Search and Optimization: Estimation, Simulation and Control*, Wiley, 2003.

Some examples of the SA algorithm:

a. *Average of an i.i.d. sequence:* Let $(Z_n)_{\geq 0}$ be an i.i.d. sequence with mean $\mu = E(Z_0)$ and finite variance. We wish to estimate the mean.

The iterative algorithm

$$\theta_{n+1} = \theta_n + \frac{1}{n+1}[Z_n - \theta_n]$$

gives

$$\theta_n = \frac{1}{n}\theta_0 + \frac{1}{n}\sum_{k=0}^{n-1} Z_k \to \mu \quad \text{(w.p. 1)}, \quad \text{by the SLLN.}$$

This is a SA iteration, with $\alpha_n = \frac{1}{n+1}$, and $Y_n = Z_n - \theta_n$. Writing $Z_n = \mu + \omega_n$ (Z_n is considered a noisy measurement of μ, with zero-mean noise ω_n), we can identify $h(\theta) = \mu - \theta$.

b. *Function minimization:* Suppose we wish to minimize a (convex) function $f(\theta)$. Denoting $h(\theta) = -\nabla f(\theta) \equiv -\frac{\partial f}{\partial \theta}$, we need to solve $h(\theta) = 0$.

The basic iteration here is

$$\theta_{n+1} = \theta_n + \alpha_n[-\nabla f(\theta) + \omega_n].$$

This is a "noisy" gradient descent algorithm.

When ∇f is not computable, it may be approximated by finite differences of the form

$$\frac{\partial f(\theta)}{\partial \theta_i} \approx \frac{f(\theta + e_i\delta_i) - f(\theta - e_i\delta_i)}{2\delta_i}\,.$$

where e_i is the i-th unit vector. This scheme is known as the "Kiefer-Wolfowitz Procedure".

Some variants of the SA algorithm

- *A fixed-point formulation:* Let $h(\theta) = H(\theta) - \theta$. Then $h(\theta) = 0$ is equivalent to the fixed-point equation $H(\theta) = \theta$, and the algorithm is

$$\theta_{n+1} = \theta_n + \alpha_n[H(\theta_n) - \theta_n + \omega_n] = (1 - \alpha_n)\theta_n + \alpha_n[H(\theta_n) + \omega_n]\,.$$

  This is the form used in the Bertsekas & Tsitsiklis (1996) monograph.

  Note that in the average estimation problem (example a. above) we get $H(\theta) = \mu$, hence $Z_n = H(\theta_n) + \omega_n$.

- *Asynchronous updates:* Different components of θ may be updated at different times and rates. A general form of the algorithm is:

$$\theta_{n+1}(i) = \theta_n(i) + \alpha_n(i)Y_n(i), \quad i = 1, \cdots, d$$

where each component of θ is updated with a different gain sequence $\{\alpha_n(i)\}$. These gain sequences are typically required to be of comparable magnitude.

Moreover, the gain sequences may be allowed to be *stochastic*, namely depend on the entire history of the process up to the time of update. For example, in the TD(0) algorithm θ corresponds to the estimated value function $\hat{V} = (\hat{V}(s),\, s \in S)$, and we can define $\alpha_n(s) = 1/N_n(s)$, where $N_n(s)$ is the number of visits to state s up to time n.

- *Projections:* If is often known that the required parameter θ lies in some set $B \subset \mathbb{R}^d$. In that case we could use the projected iterates:

$$\theta_{n+1} = Proj_B[\theta_n + \alpha_n Y_n]$$

where $Proj_B$ is some projection onto B.

The simplest case is of course when B is a box, so that the components of θ are simply truncated at their minimal and maximal values.

If B is a bounded set then the estimated sequence $\{\theta_n\}$ is guaranteed to be bounded in this algorithm. This is very helpful for convergence analysis.

7.3 Assumptions

Gain assumptions To obtain convergence, the gain sequence needs to decrease to zero. The following assumption is standard.

Assumption G1: $\alpha_n \geq 0$, and

$$\text{(i)} \quad \sum_{n=1}^{\infty} \alpha_n = \infty$$

$$\text{(ii)} \quad \sum_{n=1}^{\infty} \alpha_n^2 < \infty\,.$$

A common example is $\alpha_n = \dfrac{1}{n^a}$, with $\frac{1}{2} < a \leq 1$.

Noise Assumptions In general the noise sequence $\{\omega_n\}$ is required to be "zero-mean", so that it will average out.

Since we want to allow dependence of ω_n on θ_n, the sequence $\{\omega_n\}$ cannot be assumed independent. The assumption below allows $\{\omega_n\}$ to be a martingale difference sequence.

Let

$$\mathcal{F}_{n-1} = \sigma\{\theta_0, \alpha_0, \omega_0, \cdots, \omega_{n-1}; \theta_n, \alpha_n\}$$

denote the (σ-algebra generated by) the history sequence up to step n. Note that ω_n is measurable on $\mathcal{F}_n$ by definition of the latter.

Assumption N1:

(a) The noise sequence $\{\omega_n\}$ is a martingale difference sequence relative to the filtration $\{F_n\}$, namely

$$E(\omega_n|\mathcal{F}_{n-1}) = 0 \qquad \text{(a.s.)}.$$

(b) For some finite constants A, B and some norm $\|\cdot\|$ on $\mathbb{R}^d$,

$$E(\|\omega_n\|^2|\mathcal{F}_{n-1}) \le A + B\|\theta_n\|^2 \quad \text{(a.s.)}, \quad \forall n \ge 1\,.$$

Example: Let $\omega_n \sim N(0, \sigma_n)$, where σ_n may depend on θ_n, namely $\sigma_n = f(\theta_n)$. Formally,

$$E(\omega_n|F_n) = 0$$
$$E(\omega_n^2|F_n) = f(\theta_n)^2,$$

and we require that $f(\theta)^2 \le A + B\theta^2$.

Note: When $\{\theta_n\}$ is known to be bounded, then (b) reduces to

$$E(\|\omega_n\|^2|\mathcal{F}_{n-1}) \le C \quad \text{(a.s.)} \quad \forall n$$

for some $C < \infty$. It then follows by the martingale difference convergence theorem that

$$\lim_{n\to\infty} \frac{1}{n}\sum_{k=1}^{n} \omega_k = 0 \quad \text{(a.s.)}.$$

However, it is often the case that θ is not known to be bounded *a-priori*.

Markov Noise: The SA algorithm may converge under more general noise assumptions, which are sometimes useful. For example, for each fixed θ, ω_n may be a *Markov chain* such that its long-term average is zero (but $E(\omega_n|\mathcal{F}_{n-1}) \neq 0$). We shall not go into that generality here.

7.4 The ODE Method

The asymptotic behavior of the SA algorithm is closely related to the solutions of a certain ODE (Ordinary Differential Equation), namely

$$\frac{d}{dt}\theta(t) = h(\theta(t)),$$

or $\dot{\theta} = h(\theta)$.

Given $\{\theta_n, \alpha_n\}$, we define a *continuous-time* process $\theta(t)$ as follows. Let

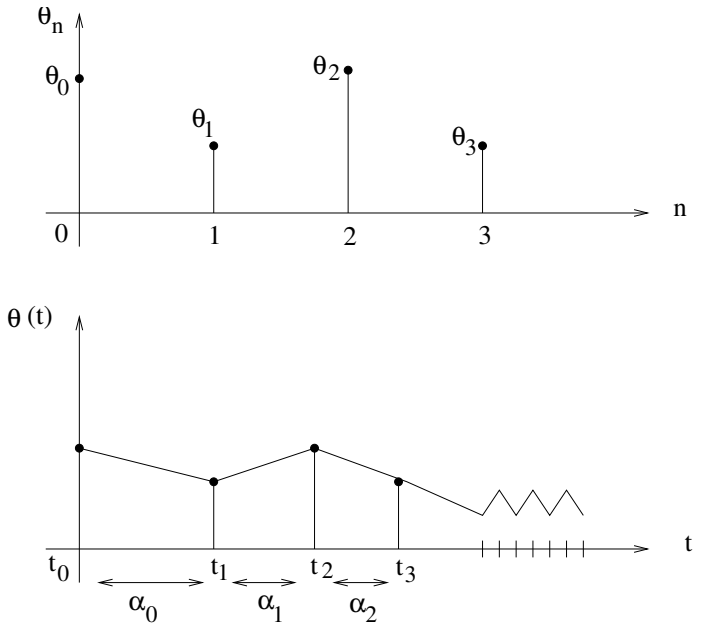
$$t_n = \sum_{k=0}^{n-1} \alpha_k \, .$$

Define

$$\theta(t_n) = \theta_n \, ,$$

and use linear interpolation in-between the t_n's.

Thus, the time-axis t is rescaled according to the gains $\{\alpha_n\}$.



Note that over a fixed Δt, the "total gain" is approximately constant:

$$\sum_{k \in K(t,\Delta t)} \alpha_k \simeq \Delta t \, ,$$

where $K(t, \Delta t) = \{k : t \leq t_k < t + \Delta t\}$.

Now:

$$\theta(t + \Delta t) = \theta(t) + \sum_{k \in k(t,\Delta t)} \alpha_k [h(\theta_n) + \omega_n] \, .$$

- For t large, α_k becomes small and the summation is over many terms; thus the noise term is approximately "averaged out": $\sum \alpha_k \omega_k \to 0$.
- For Δt small, θ_k is approximately constant over $K(t, \Delta t) : h(\theta_k) \simeq h(\theta(t))$.

We thus obtain:

$$\theta(t + \Delta t) \simeq \theta(t) + \Delta t \cdot h(\theta(t)) \,.$$

For $\Delta t \to 0$, this reduces to the ODE:

$$\dot{\theta}(t) = h(\theta(t)) \,.$$

To conclude:

- As $n \to \infty$, we "expect" that the estimates $\{\theta_n\}$ will follow a trajectory of the ODE $\dot{\theta} = h(\theta)$ (under the above time normalization).
- Note that the stationary point(s) of the ODE are given by $\theta^* : \quad h(\theta^*) = 0$.
- An obvious requirement for $\theta_n \to \theta^*$ is $\theta(t) \to \theta^*$ (for any $\theta(0)$). That is: θ^* is a *globally asymptotically stable* equilibrium of the ODE.

  This may this be viewed as a necessary condition for convergence of θ_n. It is also sufficient under additional assumptions on h (continuity, smoothness), and boundedness of $\{\theta_n\}$.

7.5 Some Convergence Results

A typical convergence result for the (synchronous) SA algorithm is the following:

Theorem 7.5. *Assume G1, N1, and furthermore:*

(i) h is Lipschitz continuous.

(ii) The ODE $\dot{\theta} = h(\theta)$ has a unique equilibrium point θ^*, which is globally asymptotically stable.

(iii) The sequence (θ_n) is bounded (with probability 1).

Then $\theta_n \to \theta^*$ (w.p. 1), for any initial conditions θ_0.

Remarks:

1. More generally, even if the ODE is not globally stable, θ_n can be shown to converge to an *invariant set* of the ODE (e.g., a limit cycle).
2. Corresponding results exist for the asynchronous versions, under suitable assumptions on the relative gains.
3. A major assumption in the last result in the boundedness of (θ_n). In general this assumption has to be verified independently. However, there exist several results that rely on further properties of h to deduce boundedness, and hence convergence.

The following convergence result from B. &T. (1996) relies on contraction properties of H, and applies to the asynchronous case. It will directly apply to some of our learning algorithms. We start with a few definitions.

- Let $H(\theta) = h(\theta) + \theta$, so that $h(\theta) = H(\theta) - \theta$.
- Recall that $H(\theta)$ is a *contraction operator* w.r.t. a norm $\|\cdot\|$ if

  $$\|H(\theta_1) - H(\theta_2)\| \leq \alpha \|\theta_1 - \theta_2\|$$

  for some $\alpha < 1$ and all θ_1, θ_2.
- $H(\theta)$ is a *pseudo-contraction* if the same holds for a fixed $\theta_2 = \theta^*$. It easily follows then that θ^* is a unique fixed point of H.
- Recall that the *max-norm* is given by $\|\theta\|_\infty = \max_i |\theta(i)|$. The *weighted* max-norm, with a weight vector w, $w(i) > 0$, is given by

  $$\|\theta\|_w = \max_i \{\frac{|\theta(i)|}{w(i)}\}\,.$$

Theorem 7.6 (Prop. 4.4. in B.&T.)**.** *Let*

$$\theta_{n+1}(i) = \theta_n(i) + \alpha_n(i)[H(\theta_n) - \theta_n + \omega_n]_i\,, \quad i = 1, \cdots, d\,.$$

Assume N1, and:

(a) Gain assumption: $\alpha_n(i) \geq 0$, measurable on the "past", and satisfy

$$\sum_n \alpha_n(i) = \infty, \quad \sum_n \alpha_n(i)^2 < \infty \quad \textit{(w.p. 1)}\,.$$

(b) H is a pseudo-contraction w.r.t. some weighted max-norm.

Then $\theta_n \to \theta^*$ (w.p. 1), where θ^* is the unique fixed point of H.

Remark on "Constant Gain" Algorithms As noted before, in practice it is often desirable to keep a non-diminishing gain. A typical case is $\alpha_n(i) \in [\underline{\alpha}, \overline{\alpha}]$.

Here we can no longer expect "w.p. 1" convergence results. What can be expected is a statement of the form:

- For $\overline{\alpha}$ small enough, we have for all $\epsilon > 0$

  $$\limsup_{n\to\infty} P(\|\theta_n - \theta^*\| > \epsilon) \leq b(\epsilon) \cdot \overline{\alpha}\,,$$

  with $b(\epsilon) < \infty$.

This is related to "convergence in probability", or "weak convergence". We shall not give a detailed account here.

7.6 Exercises

Exercise 7.1 (**Stochastic Approximation**)**.** Let $\{X_n\}_{n=0}^{\infty}$ be an i.i.d. sequence of bounded random variables with mean μ and variance σ^2. Consider the following iterative method for estimating μ, σ^2:

$$\begin{aligned}
\hat{\mu}_0 &= \hat{s}_0 = 0, \\
\hat{\mu}_{n+1} &= \hat{\mu}_n + \alpha_n \left(X_{n+1} - \hat{\mu}_n \right), \\
\hat{s}_{n+1} &= \hat{s}_n + \alpha_n \left((X_{n+1} - \hat{\mu}_n)^2 - \hat{s}_n \right).
\end{aligned}$$

Use the ODE method and show that $(\hat{\mu}_n, \hat{s}_n) \to \left(\mu, \sigma^2\right)$ with probability one.

Chapter 8

Basic Convergence Results for RL Algorithms

We establish here some asymptotic convergence results for the basic RL algorithms, by showing that they reduce to Stochastic Approximation schemes. We focus on the discounted-cost problem, which is easiest. Analogous results exist for shortest-path problems under the properness assumption (every policy terminates in expected finite time).

We do not directly consider here the issue of *exploration*, which is essential for convergence to the optimal policy. Thus, where required we will simply *assume* that all actions are sampled often enough.

8.1 Q-learning

Recall the Q-learning algorithm, in the following generalized form:

$$Q_{n+1}(s,a) = Q_n(s,a) + \alpha_n(s,a)[r(s,a,s'_{s,a}) + \gamma \max_{a'} Q_n(s'_{s,a}, a') - Q_n(s,a)]$$

where each $s'_{s,a}$ is determined randomly according to $p(s'|s,a)$.

We allow here $\alpha_n(s,a) = 0$, so that any number of (s,a) pairs can be updated at each stage.

This iteration can be viewed as an (asynchronous) Stochastic Approximation algorithm, with $Q \equiv \theta$. This leads to the following result.

Theorem 8.1 (Convergence of Q-learning.)**.** *Let $\gamma < 1$, and let Q^* be the optimal γ-discounted Q function. Assume*

$$\sum_{h=0}^{\infty} \alpha_n(s,a) = \infty, \quad \sum_{n=0}^{\infty} \alpha_n(s,a)^2 < \infty \quad \textit{(w.p. 1)} \quad \forall s, a\,.$$

Then

$$\lim_{n\to\infty} Q_n(s,a) = Q^*(s,a) \quad \textit{(w.p. 1)} \quad \forall s,a\,.$$

Proof. Define the mapping H over the set of Q-functions as follows:

$$\begin{aligned}(HQ)(s,a) &= \sum_{s'} P(s'|s,a)[r(s,a,s') + \gamma \max_{a'} Q(s',a')] \\ &= E[r(s,a,s_{n+1}) + \gamma \max_{a'} Q(s_{n+1},a')|s_n = s, a_n = a]\,.\end{aligned}$$

The above Q-learning algorithm can thus be written in the standard SA form, with the noise vector ω_n given by:

$$\omega_n(s,a) = r(s,a,s'_{s,a}) + \gamma \max_{a'} Q_n(s'_{s,a},a') - (HQ)(s,a)\,.$$

We proceed to verify the assumptions in Theorem 7.6:

(a) Step-size requirements hold here by assumption.

(b) Noise Assumption N1: The definition of ω_n immediately implies that $E(\omega_n(s,a)|\mathcal{F}_n) = 0$. It is further easily seen that

$$E(\omega_n(s,a)^2|\mathcal{F}_n) \leq \text{ quadratic function of } \|Q\|_\infty\,.$$

(c) Contraction: As with the discounted DP operator, it may be verified that H is a γ-contraction w.r.t. the max-norm.

The required convergence result therefore follows by Theorem 7.6. □

Remarks on basic (on-policy) Q-learning:

- In the basic version of the algorithm, we follow a state-action sequence $(s_n, a_n; n = 0, 1, \cdots)$ which is generated by some arbitrary policy, and at time n update $Q(s,a)$ only for $(s,a) = (s_n, a_n)$. This corresponds to the choice of gains:

$$\alpha_n(s,a) > 0 \quad \text{iff} \quad (s,a) = (s_n, a_n)\,.$$

- For $(s,a) = (s_n, a_n)$, a typical choice for α_n is

$$\alpha_n(s,a) = \hat{\alpha}(N_n(s,a))$$

  where N_n is the number of previous visits to (s,a), and $\hat{\alpha}(k)$ satisfies the standard assumptions.

- For the step-size requirements in the theorem to hold in this case it is required that each (s,a) pair is visited "relatively often". This should be verified by appropriate exploration policies!

Undiscounted case: Under appropriate "Stochastic Shortest Path" assumptions, it can be shown that H is a pseudo-contraction w.r.t. some weighted max-norm. Convergence follows as above.

8.2 Convergence of TD(λ)

TD(0) can be analyzed exactly as Q-learning learning. TD(λ) is slightly more involved. Recall the "on-line" version of TD(λ):

$$V_{n+1}(s) = V_n(s) + \alpha_n e_n(s) d_n\,, \quad s \in S$$

where

$$\begin{aligned} \alpha_n &= \text{gain} \\ e_n(s) &= \text{eligibility trace coefficient} \\ d_n &= r_n + \gamma V_n(s_{n+1}) - V_n(s_n) \\ \gamma &= \text{discount factor} \end{aligned}$$

Requirements on the Eligibility Trace: Several variants of the algorithm are obtained by different choices of $e_n(s)$, such as:

(a) First-visit TD(λ):

$$e_n(s) = (\gamma\lambda)^{n-m_1(s)} 1\{n \geq m_1(s)\}\,,$$

$m_1(s)$ is the time of first visit to state s (during the present run).

(b) Every-visit TD(λ):

$$e_n(s) = \sum_{j:m_j(s)\leq n} (\gamma\lambda)^{n-m_j(s)}\,,$$

$m_j(s)$ is the time of j^{th} visit to state s.

(c) First-visit with stopping:

$$e_n(s) = (\gamma\lambda)^{n-m_1(s)} 1\{m_1(s) \leq n \leq \tau\}$$

where τ is some stopping time – e.g., end of simulation run, or arrival to a state whose value $V(s)$ is known with high precision. $e_n(s)$ is restarted after τ.

A *general set of requirements* on the eligibility coefficients $e_n(s)$, which includes the above cases, is given as follows:

(a) $e_0(s) = 0$, $e_n(s) \geq 0$.

(b) $e_n(s) \leq \gamma e_{n-1}(s)$ if $s_n \neq s$,
$1 \leq e_n(s) \leq 1 + \gamma e_{n-1}(s)$ if $s_n = s$.

(c) $e_n(s)$ is measurable on the past.

Convergence: We now argue as follows.

- It may be seen that TD(λ) is in the form of the Stochastic Approximation algorithm, with $\theta_n \equiv V_n$, and

$$h(\theta) \equiv h(V) = (h(V)(s),\, s \in S)\,,$$

$$\begin{aligned} h(V)(x) &= E^{\pi}[d_n | V_n = V,\, s_n = s] \\ &= \sum_a \pi(a|s)[r(s,a) + \gamma \sum_{s'} p(s'|s,a)V(s')] - V(s) \\ &:= (HV)(s) - V(s)\,. \end{aligned}$$

  Here π is the *fixed stationary* policy that is used.

- For $0 < \gamma < 1$ it is obvious that H is a contraction operator.

- For convergence we now need to verify that the effective gains $\alpha_n e_n(s)$ satisfy the "usual assumptions". This may be verified by requiring that each state is visited "relatively often".

For $\gamma = 1$, a similar argument may be made for SSP (Stochastic Shortest Path) problems.

8.3 Actor-Critic Algorithms

Convergence of actor-critic type algorithms is harder to analyze. We describe here some results from Konda and Borkar (2000).

Recall that the idea is to use a "fast" estimation loop to obtain $\hat{V}(s)$, and a slower loop to update the policy $\hat{\pi}$ given $\hat{V}$.

Let $V_n(s)$ and $\pi_n = (\pi_n(a|s))$ be the estimated value and policy at step n.

Algorithm 1

a. Value-function estimation (generalized TD(0)):

$$V_{n+1} = V_n(s) + \beta_n(s)[r(s, a_n(s)) + \gamma V_n(s_{n+1}(s)) - V_n(s)]\,, \quad s \in Y_n$$

where

Y_n – set of states updated at step n

$\beta_n(s)$ – gains

$s_{n+1}(s)$ – next state, chosen with distribution $p(s') = \sum_a p(s'|s,a)\pi_n(a,s)$.

b. Policy update:

$$\pi_{n+1}(a|s) = \pi_n(a|s) + \alpha_n(s,a)((\hat{Q}_n(s,a) - \hat{Q}_n(s,a_0))), \quad (s,a) \in Z_n$$

where

Z_n – set of state-action pairs updated at step n
$\alpha_n(s,a)$ – gains
$\hat{Q}_n(s,a) := r(s,a) + \gamma V_n(s_{n+1}(s,a))$
$s_{n+1}(s,a)$ – next state, chosen according to $p(s'|s,a)$
a_0 – a fixed reference action (for each state).

b'. Policy normalization:

For each s, project the vector $(\pi_{n+1}(a|s),\ a \neq a_0)$ unto the following set of sub-probability vectors:

$$\{\pi : \pi(a) \geq 0,\ \sum_{a \neq a_0} \pi(a) \leq 1\}$$

and then let $\pi_{n+1}(a_0|s) = 1 - \sum_{a \neq a_0} \pi_{n+1}(a|s)$.

c. Rate requirements:

We first require that all updates are executed relatively often, namely that for some $\Delta > 0$,

$$\liminf_{n\to\infty} \frac{n_1(s)}{n} \geq \Delta\,, \quad \liminf_{n\to\infty} \frac{n_2(s,a)}{n} \geq \Delta\,,$$

where

$$n_1(s) = \sum_{k=1}^{n} 1\{s \in Y_k\}$$

$$n_2(s,a) = \sum_{k=1}^{n} 1\{(s,a) \in Z_k\}\,.$$

The gains are determined by some sequences $\alpha(m)$ and $\beta(m)$, as

$$\alpha_n(s,a) = \alpha(n_2(s,a))\,, \quad \beta_n(s) = \beta(n_1(s))\,.$$

The sequences $\alpha(m)$, $\beta(m)$ should satisfy:

(1) The standard summability assumptions.

(2) Policy updates are "slower": $\lim_{m\to\infty} \frac{\alpha(m)}{\beta(m)} = 0$.

(3) Some additional technical assumptions ...

All these requirements are satisfied, e.g., by $\alpha(m) = \frac{1}{m \log m}$, $\beta(m) = \frac{1}{m}$.

Under these assumptions, Algorithm 1 converges to the optimal value and policy.

Algorithm 2: Same as Algorithm 1, except for the policy update (b):

$$\pi_{n+1}(a|s) = \pi_n(a|s) + a_n(s,a)[\{\hat{Q}_n(s,a) - V_n(s)\}\pi_n(a|s) + \xi_n(s,a)]\,.$$

$\xi_n(s,a)$ are sequences of "small" noise terms, these are needed to prevent the algorithm from getting stuck in the wrong "corners".

Algorithm 3: Same as Algorithm 1, except for (b):

$$w_{n+1}(a|s) = w_n(a|s) + \alpha_n(s,a)[\hat{Q}_n(s,a) - V_n(s)]$$

and

$$\pi_n(s,a) := \frac{\exp(w_n(s,a))}{\sum_{a'}\exp(w_n(s,a'))}\,.$$

In all these variants, convergence is proved using a "two-time scale" Stochastic Approximation framework, the analysis is based on the ODE method which couples a "fast" ODE (for V) and a "slow" ODE (for π).

Chapter 9

Approximate Dynamic Programming

Recall Bellman's dynamic programming equation

$$V(s) = \max_{a \in A} \left\{ R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V(s') \right\},$$

or in operator form, $\underline{V} = T\underline{V}$, where T is the Bellman operator.

Dynamic programming requires knowing the model and is only feasible for small problems. In large scale problems, we face the **3 curses of dimensionality**:

1. S may be large, such that even writing down the policy is difficult. Moreover, S may be continuous, for example in robotics applications.
2. A may be large. Example: resource allocation, where we have several projects and need to assign different resources to each project.
3. $p(s'|s,a)$ may be complicated: computing $p(s'|s,a)$ requires summing over many random events. Example: resource allocation.

9.1 Approximation approaches

There are 4 approaches to handle the curses of dimensionality:

1. **Myopic**: When $p(s'|s,a)$ is approximately uniform across a, we may ignore the state transition dynamics and simply use $a^\pi(s) \approx \underset{a \in A}{\text{argmax}}\{R(s,a)\}$. If $R(s,a)$ is not known exactly – replace it with an estimate.

2. **Lookahead policies**: Rolling horizon/model-predictive control.
   Simulate a horizon of T steps, and use

$$a^{\pi}(s_t|\theta) = \underset{\pi' \in \Pi}{\operatorname{argmax}} \, \mathbb{E}\left[\sum_{t'=t}^{t+T} R(s_{t'}, y^{\pi'}(s_{t'}))\right]$$

3. **Policy function approximation**
   Assume policy is of some parametric function form $\pi = \pi(\theta), \quad \theta \in \Theta$, and optimize over function parameters.

Example 9.1 (Inventory management)**.** *Consider an inventory management problem, in which the state is the inventory level s_t, the demand is d_t and the action is the replenishment level a_t. The dynamics are given by:*

$$s_{t+1} = [s_t + a_t - d_t]^+.$$

The immediate reward is:

$$R_t = R \min\{d_t, s_t + a_t\} - c_a a_t - C[s_t + a_t - d_t]^+,$$

where R is the profit in satisfying the demand, c_a is the ordering cost, and C is the cost of not satisfying a demand.

One possible replenishment policy is:

$$a^{\pi}(s_t|\theta) = \begin{cases} 0 & s_t > q \\ Q - s_t & s_t \le q \end{cases}, \quad \theta = (q, Q).$$

A different possibility is the softmax *policy, given by*

$$p(s,a) = \frac{e^{-\theta\alpha(s,a)}}{\sum_{a'} e^{-\theta\alpha(s,a')}}, \quad \text{where} \quad \alpha = R(s,a) \quad \text{or} \quad \alpha = Q(s,a).$$

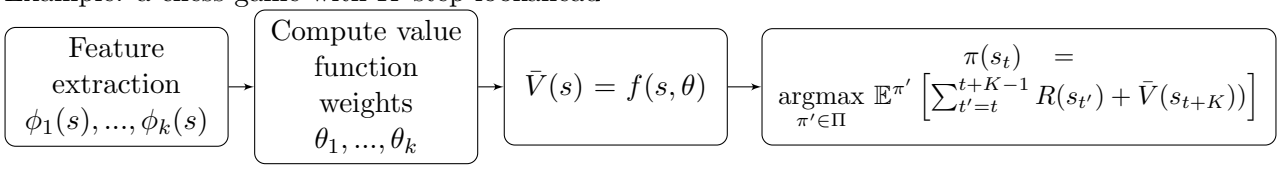
4. **Value function approximation**
   Assume $V(s) \approx f(s,\theta)$ where f is some approximation function, and optimize over the parameters θ.
   The most standard model is a linear combination of some k features (which are not necessarily linear):

$$\bar{V}(s|\theta) = \sum_{j=1}^{k} \theta_j \phi_j(s),$$

   where θ_j are the model parameters and ϕ_j are the model's *features* (a.k.a. basis functions).

Given $\bar{V}$, we can derive a policy by choosing the *greedy* action with respect to $\bar{V}$ (corresponding to a lookahead of 1 time step). If a model or a simulator is also available, we can also use a longer lookahead.

Example: a chess game with K-step lookahead



Issues:

(a) Choosing the parametric class ϕ (architecture): e.g., Radial Basis Functions (RBFs) or kernels. The difficulty is that the value function structure may be hard to know in advance.

(b) Tuning the weights θ (learning): here we focus on simulation.
In the linear architecture, we can write

$$\tilde{V}(s; \theta) = \phi(s)^\top \theta$$

(replacing θ with r for the parameters), or in matrix-vector form,

$$\tilde{V}(\theta) = \Phi\theta$$

Therefore tuning the parameters θ reduces to approximating $V \in \mathbb{R}^{|S|}$ on $\mathcal{S} = \{\Phi\theta : \theta \in \mathbb{R}^k\} \subseteq \mathbb{R}^{|S|}$.

9.2 Lookahead Policies

We will discuss several variants of lookahead policies. We focus on systems with a discrete, and 'small enough' action space, but a large, possibly infinite state space.

The main idea in lookahead policies is that when we are at some state s, we will only search 'around' the state s for the next action $\pi(s)$. After we play the action, we observe the next state, and repeat the search. Thus, we do not need to consider the whole state space in advance, but only consider regions in state space which we actually visit. In the control literature, this idea is often called Model Predictive Control (MPC).

9.2.1 Deterministic Systems: Tree Search

The simplest example of a lookahead policy is tree search in a deterministic system.

Given s, and a (deterministic) simulator of the system, we can simulate the next state for every possible action, building a tree of possible trajectories of the system, starting from s, and continuing to some depth k. We then choose an action by:

$$\pi(s) = \operatorname*{argmax}_{\pi' \in \Pi} \left[\sum_{t=0}^{k} \gamma^t r(s_t, \pi'(s_t)) \middle| s_0 = s \right],$$

Which can be solved using dynamic programming.

In this approach, the computation in each step requires $O(|A|^k)$ calls to the simulator for building the tree, and $O(k|A|^k)$ computations for searching the tree. Note that *there is no dependence on* $|S|$. The depth k should be chosen 'deep enough' to results in meaningful actions. For example, in a discounted setting, recall that the k-horizon value function satisfies $\|V^k(s) - V^*(s)\|_\infty \leq \gamma^k \left(\frac{R_{max}}{1-\gamma} \right)$, which can be used to set k.

9.2.2 Stochastic Systems: Sparse Sampling

When the system is stochastic, we cannot simply build a tree of possible future trajectories. If we sample next states from our simulator, we can build a *sampled* version of a search tree. Here, for each state s and action a in the tree, we will sample C next states $s'_1, \ldots, s'_C$. Then, the backup at s, a will be $Q(s,a) = \frac{1}{C} \sum_{i=1}^{C} r(s,a) + \gamma \max_{a'} Q(s'_i, a')$. That is, we replaced the expectation in the Bellman update with an empirical average. Since the average concentrates around the mean (e.g., by Hoeffding inequality), it is expected that the empirical averages will closely represent their expectations.

The following result is shown in [1]. The constants k and C can be set such that with a per-state running time of $(\frac{|A|}{\epsilon(1-\gamma)})^{O\left(\frac{1}{1-\gamma} \log \frac{1}{\epsilon(1-\gamma)}\right)}$, the obtained policy is $\epsilon-$optimal, i.e., $\|V^k(s) - V^*(s)\|_\infty \leq \epsilon$. Note again that there is no dependence on $|S|$.

9.2.3 Monte Carlo Tree Search

While the sparse sampling above does not depend on $|S|$, the exponential dependence on the horizon and number of actions makes it impractical for many cases. The insight in Monte-Carlo Tree Search (MCTS) is that investing the same simulation efforts for all possible future actions is wasteful, and we should instead focus our search efforts on the *most promising* future outcomes.

The first idea in MCTS is to replace the stage-wise tree building approach with a rollout-based approach, as shown in Figure 9.1. Here, we roll out complete trajectories of depth k, and build the tree progressively from these rollouts. The method **Evaluate(state)** is used to evaluate the last state in the rollout. This could be the reward of the state, or an estimated value function (as in the chess example above). Another approach, which has become common in games such as Go and Chess, is to simulate a game starting from state s with a predetermined policy, until termination, and evaluate the state by the empirical

return along the simulation trajectory. The method **UpdateValue** is used to build the tree. It maintains both the sum of returns from a state action pair $Q_{sum}(s,a)$ (summing over all trajectories that visited this state), and the number of visits to the state and state-action pair, N_s and N_{sa}. The value of a state-action pair is the average $\frac{Q_{sum}(s,a)}{N_{sa}}$. To understand why this approach could be beneficial, consider a state that is visited multiple times. Thus, after several visits, we have some information about which actions are more promising, and could use that to bias the rollout policy to focus on promising trajectories. We need to make sure, however, that we also *explore* often enough, so that we do not miss important actions by incorrect Q estimates in the beginning. The key to an efficient MCTS algorithm, therefore, is in the **selectAction** method, which should balance exploration and exploitation.

```
 1: function MonteCarloPlanning(state)
 2: repeat
 3:     search(state, 0)
 4: until Timeout
 5: return bestAction(state,0)

 6: function search(state, depth)
 7: if Terminal(state) then return 0
 8: if Leaf(state, d) then return Evaluate(state)
 9: action := selectAction(state, depth)
10: (nextstate, reward) := simulateAction(state, action)
11: q := reward + γ search(nextstate, depth + 1)
12: UpdateValue(state, action, q, depth)
13: return q
```

Figure 9.1: Generic Monte-Carlo Planning Algorithm (from[2])

The UCT algorithm of Kocsis and Szepesvari [2] selects actions that maximize $Q(s,a)+UCT(s,a)$, where the *upper confidence bound* is given by:

$$UCT(s,a) = C\sqrt{\frac{\log N_s}{N_{sa}}},$$

where C is some constant. Intuitively, UCT prefers actions that are either promising (high Q) or under-explored (low N_{sa}). Later in the course we will show that this specific formulation is actually optimal in some sense.

In many practical problems, UCT has been shown to be much more efficient than tree-building methods. MCTS based on UCT is also the most prominent method for solving games with high branching factors such as Go.

9.3 Approximation methods in value space

1. Approximate policy iteration: approximate V^μ; improve μ; repeat.
2. Approximate value iteration / Q-learning: $\hat{Q}(s,a) \approx \phi(s,a)^\top \theta$
3. Linear programming: not discussed.

Note that both V and Q are mappings from a state (or state-action) to $\mathbb{R}$. When the state space is large, we will not be able to calculate the value for every state, but instead, fit some function to a few states values that we will calculate. This is similar to a *regression* problem, which our development will be based on, but as we will see, we will also need to account for the dynamic nature of the problem to develop approximate optimization algorithms.

9.3.1 Approximate Policy Evaluation

We start with the simplest setting - approximating the value of a fixed policy. A direct approach for this task is through regression, which we will now review.

Least Squares Regression

Assume we have some function $y = f(x)$, and a distribution over a finite set of inputs $\xi(x)$. We want to fit a parametric function $g(x;\theta)$ to our data such that g approximates f. The least squares approach solves the following problem:

$$\min_\theta \mathbb{E}_{x\sim\xi}(g(x;\theta) - f(x))^2.$$

When g is linear in some features $\phi(x)$, i.e., $g(x;\theta) = \theta^T\phi(x)$, then we can write the above equation in vector notation as

$$\min_\theta (\Phi\theta - Y)^T \Xi (\Phi\theta - Y),$$

where Y is a vector of $f(x)$ for every x, and Φ is a matrix with $\phi(x)$ as its rows. The solution is given by:

$$\theta_{LS} = (\Phi^T\Xi\Phi)^{-1}\Phi^T\Xi Y.$$

Note that $\Phi\theta_{LS}$ denotes the approximated function $g(x;\theta)$. We will call this the *projection* of y onto the space spanned by $\theta^T\phi(x)$, and we can write the projection operator explicitly as:

$$\Pi_\epsilon Y = \Phi\theta_{LS}(Y) = \Phi(\Phi^T\Xi\Phi)^{-1}\Phi^T\Xi Y.$$

In the non-linear case, a common approach is to solve the least squares problem by gradient descent, i.e., $\theta_{k+1} = \theta_k - \alpha_k \nabla_\theta \mathbb{E}_{x\sim\xi}(g(x;\theta) - f(x))^2$, where the gradient is simply $2\mathbb{E}_{x\sim\xi}(g(x;\theta) - f(x))\nabla_\theta g(x;\theta)$.

In a practical case, we are given N data samples $\{x_i, y_i\} : x_i \sim \xi, y_i = f(x_i)$. The least squares solution can be approximated from the samples as:

$$\min_\theta \sum_i (g(x_i;\theta) - y_i)^2,$$

The stochastic gradient descent (SGD) version of this update is

$$\theta_{k+1} = \theta_k - \alpha_k (g(x_i;\theta_k) - y_i) \nabla_\theta g(x_i;\theta_k).$$

For the linear case, we can solve directly:

$$\hat{\theta}_{LS} = (\hat{\Phi}^T \hat{\Phi})^{-1} \hat{\Phi}^T \hat{Y},$$

where in this case the rows of $\hat{\Phi}$ and $\hat{Y}$ are given by the N samples. By the law of large numbers, we have that $\frac{1}{N}(\hat{\Phi}^T \hat{\Phi}) \to (\Phi^T \Xi \Phi)$ and $\frac{1}{N} \hat{\Phi}^T \hat{Y} \to \Phi^T \Xi Y$, thus the sampled solution converges to the solution θ_{LS} above.

Approximate Policy Evaluation: Regression

The direct approach to policy evaluation: Evaluate $V^\mu(s)$ only for several states (e.g. by simulation) and interpolate, e.g., using least squares regression: $\Phi\theta = \Pi V^\mu$. This method is simple, however, it has several drawbacks. The first is that value estimates typically have a large variance, as they accumulate rewards from multiple states in the future. The second is that we need to wait for full simulations to complete before updating the value estimate. Practically, this can take too long in some problems.

We next describe an alternative approach, using the fact that the value function satisfies Bellman's equation.

Approximate Policy Evaluation: the Projected Bellman Equation

The indirect approach to policy evaluation is based on the projected Bellman equation (PBE). Recall that $V^\mu(s)$ satisfies the Bellman equation: $V^\mu = T^\mu V^\mu$. We can evaluate $V^\mu(s)$ by projecting the Bellman operator T^μ onto $\mathcal{S}$ (the subspace spanned by Φ):

$$\Phi\theta = \Pi T^\mu \{\Phi\theta\},$$

where Π is the projection operator onto $\mathcal{S}$ under some norm. Includes:

- Temporal differences - TD(0): online solution of the PBE.
- Least-squares policy evaluation - LSPE(0): simulation-based form

$$\Phi\theta_{k+1} = \Pi T^\mu \{\Phi\theta_k\} + \text{noise}.$$

- Least-squares temporal differences - LSTD(0): batch solution of the PBE.

We now discuss the indirect approach to policy evaluation.
Define the weighted Euclidean inner product:

$$\langle V_1, V_2 \rangle_\epsilon = \sqrt{\sum_{i=1}^{n} \epsilon_i V_1(i) V_2(i)}, \quad \epsilon_i \geq 0,$$

and the induced weighted Euclidean norm:

$$||V||_\epsilon = \sqrt{\sum_{i=1}^{n} \epsilon_i V(i)^2}, \quad \epsilon_i \geq 0,$$

and let Π_ϵ be the operator of projection onto $\mathcal{S}$ w.r.t. this norm:

$$\begin{aligned} \Pi_\epsilon V &= \operatorname*{argmin}_{V' \in \mathcal{S}} ||V' - V||_\epsilon = \\ &= \Phi\theta \quad \text{s.t.} \quad \theta = \operatorname*{argmin}_{\theta' \in \mathbb{R}^k} ||\Phi\theta' - V||_\epsilon. \end{aligned}$$

We want to solve the projected Bellman equation (PBE):

$$\Phi\theta^* = \Pi_\epsilon T^\mu \Phi\theta^*. \tag{9.1}$$

The main reason to consider the approximation in (9.1) is that, as we shall see, it will allow us to derive efficient sampling based algorithms with provable error guarantees.

9.3.2 Existence, Uniqueness and Error Bound on PBE Solution

We are interested in the following questions:

1. Does the PBE (9.1) have a solution?
2. When is $\Pi_\epsilon T^\mu$ a contraction, and what is its fixed point?
3. If $\Pi_\epsilon T^\mu$ has a fixed point θ^*, how far is it from $\Pi_\epsilon V^\mu$?

Let us assume the following:

Assumption 9.1. *The Markov chain corresponding to μ has a single recurrent class and no transient states. We further let*

$$\epsilon_j = \lim_{N \to \infty} \frac{1}{N} \sum_{t=1}^{N} p(s_t = j | s_0 = s) > 0,$$

which is the probability of being in state j when the process reaches its steady state, given $s_0 = s$.

We have the following result:

Proposition 9.1. *Under Assumption 9.1 we have that*

1. $\Pi_\epsilon T^\mu$ *is a contraction operator with modulus* γ *w.r.t.* $||\cdot||_\epsilon$.
2. *The unique fixed point* $\Phi\theta^*$ *of* $\Pi_\epsilon T^\mu$ *satisfies*

$$||V^\mu - \Phi\theta^*||_\epsilon^2 \leq \frac{1}{1-\gamma^2}||V^\mu - \Pi_\epsilon V^\mu||_\epsilon^2.$$

Proof. We begin by showing the contraction property. We use the following lemma:

Lemma 9.1. *If* P^μ *is the transition matrix induced by* μ*, then*

$$\forall z \quad ||P^\mu z||_\epsilon \leq ||z||_\epsilon.$$

Proof. Proof: Let p_{ij} be the components of P^μ. For all $z \in \mathbb{R}^n$:

$$||P^\mu z||_\epsilon^2 = \sum_i \epsilon_i \left(\sum_j p_{ij} z_j\right)^2 \underbrace{\leq}_{\text{Jensen}} \sum_i \epsilon_i \sum_j p_{ij} z_j^2 = \sum_j z_j^2 \sum_i \epsilon_i p_{ij} = ||z||_\epsilon^2,$$

where the last equality is since by definition of ϵ_i, $\sum_i \epsilon_i p_{ij} = \epsilon_j$, and $\sum_{j=1}^n \epsilon_j z_j^2 = ||z||_\epsilon^2$. $\square$

Since Π_ϵ is a projection with respect to a weighted Euclidean norm, it obeys the Pythagorian theorem:

$$\forall J \in \mathbb{R}^{|S|}, \bar{J} \in \mathcal{S}: \quad ||J - \bar{J}||_\epsilon^2 = ||J - \Pi_\epsilon J||_\epsilon^2 + ||\Pi_\epsilon J - \bar{J}||_\epsilon^2.$$

Proof:

$$||J - \bar{J}||_\epsilon^2 = ||J - \Pi_\epsilon J + \Pi_\epsilon J - \bar{J}||_\epsilon^2 = ||J - \Pi_\epsilon J||_\epsilon^2 + ||\Pi_\epsilon J - \bar{J}||_\epsilon^2 + \langle J - \Pi_\epsilon J, \Pi_\epsilon J - \bar{J}\rangle_\epsilon,$$

and $J - \Pi_\epsilon J$ and $\Pi_\epsilon J - \bar{J}$ are orthogonal under $\langle\cdot,\cdot\rangle_\epsilon$ (error orthogonality for weighted Euclidean-norm projections).

This implies that Π_ϵ is non-expansive:

$$||\Pi_\epsilon J - \Pi_\epsilon \bar{J}||_\epsilon \leq ||J - \bar{J}||_\epsilon$$

Proof:

$$||\Pi_\epsilon J - \Pi_\epsilon \bar{J}||_\epsilon^2 = ||\Pi_\epsilon (J - \bar{J})||_\epsilon^2 \leq ||\Pi_\epsilon (J - \bar{J})||_\epsilon^2 + ||(I - \Pi_\epsilon)(J - \bar{J})||_\epsilon^2 = ||J - \bar{J}||_\epsilon^2,$$

where the first inequality is by linearity of Π_ϵ, and the last is by the Pythagorian theorem (with vectors $J - \bar{J}$ and 0).

In order to prove the contraction:

$$\begin{aligned} ||\Pi_\epsilon T^\mu J - \Pi_\epsilon T^\mu \bar{J}||_\epsilon &\overset{\Pi_\epsilon \text{ non-expansive}}{\leq} ||T^\mu J - T^\mu \bar{J}||_\epsilon \\ &\overset{\text{definition of } T^\mu}{=} \gamma ||P^\mu (J - \bar{J})||_\epsilon \overset{\text{Lemma 9.1}}{\leq} \gamma ||J - \bar{J}||_\epsilon, \end{aligned}$$

and therefore $\Pi_\epsilon T^\mu$ is a contraction operator.

Proof of the error bound:

$$\begin{aligned} ||V^\mu - \Phi\theta^*||_\epsilon^2 &= ||V^\mu - \Pi_\epsilon V^\mu||_\epsilon^2 + ||\Pi_\epsilon V^\mu - \Phi\theta^*||_\epsilon^2 \\ &= ||V^\mu - \Pi_\epsilon V^\mu||_\epsilon^2 + ||\Pi_\epsilon T^\mu V^\mu - \Pi_\epsilon T^\mu \Phi\theta^*||_\epsilon^2 \\ &\leq ||V^\mu - \Pi_\epsilon V^\mu||_\epsilon^2 + \gamma^2 ||V^\mu - \Phi\theta^*||_\epsilon^2, \end{aligned} \tag{9.2}$$

where the first equality is by the Pythagorean theorem, the second equality is since V^μ is T^μ's fixed point, and $\Phi\theta^*$ is ΠT^μ's fixed point, and the inequality is by the contraction of $\Pi_\epsilon T^\mu$.

Therefore

$$||V^\mu - \Phi\theta^*||_\epsilon^2 \leq \frac{1}{1-\gamma^2} ||V^\mu - \Pi_\epsilon V^\mu||_\epsilon^2.$$

□

Remark 9.1. *A weaker error bound of* $||V^\mu - \Phi\theta^*||_\epsilon \leq \frac{1}{1-\gamma} ||V^\mu - \Pi_\epsilon V^\mu||_\epsilon$ *may be obtained by the following argument:*

$$\begin{aligned} ||V^\mu - \Phi\theta^*||_\epsilon &\leq ||V^\mu - \Pi_\epsilon V^\mu||_\epsilon + ||\Pi_\epsilon V^\mu - \Phi\theta^*||_\epsilon \\ &= ||V^\mu - \Pi_\epsilon V^\mu||_\epsilon + ||\Pi_\epsilon T^\mu V^\mu - \Pi_\epsilon T^\mu \Phi\theta^*||_\epsilon \\ &\leq ||V^\mu - \Pi_\epsilon V^\mu||_\epsilon + \gamma ||V^\mu - \Phi\theta^*||_\epsilon, \end{aligned} \tag{9.3}$$

where the first inequality is by the triangle inequality.

Remark 9.2. *The error bounds in this section are in the* $\|\cdot\|_\epsilon$ *norm, while the approximate policy iteration error bounds of Theorem 9.1 are in the* $\|\cdot\|_\infty$ *norm. In general, for large or continuous state spaces, an* $\|\cdot\|_\epsilon$ *norm error bound does not provide a strong guarantee on the error in the* $\|\cdot\|_\infty$ *norm, and the results of Theorem 9.1 therefore do not apply. A result similar to that of Theorem 9.1 may be shown to hold in the* $\|\cdot\|_\epsilon$ *norm, however this is much more complicated, and beyond the scope of this course.*

Remark 9.3. *At this point, the reader should wonder why the PBE solution is sought instead of $\Pi_\epsilon V^\mu$. In fact, an algorithm for estimating $\Pi_\epsilon V^\mu$ can easily be derived using regression. For example, consider an algorithm that samples states $s_1, \ldots, s_N \sim \epsilon$, and from each state s_i runs a trajectory in the MDP following policy μ. Let y_i denote the discounted return in that trajectory. Then, a least squares fit:*

$$\min_\theta \frac{1}{2} \sum_{i=1}^{N} \left(y_i - \phi(s_i)^\top \theta \right)^2$$

would converge to $\Pi_\epsilon V^\mu$ as $N \to \infty$. However, such an algorithm would suffer a high variance, since the discounted return in the whole trajectory often has a high variance. The PBE approach typically has lower variance, since it only considers 1-step transitions instead of complete trajectories. However, it may have a bias*, as seen by the error bound in Proposition 9.1. We thus have a bias-variance tradeoff between the direct and indirect approaches to policy evaluation.*

9.3.3 Solving the PBE

We now move to solving the projected Bellman equation.
We would like to find $J = \Phi\theta^*$ where θ^* solves

$$\theta^* = \underset{\theta \in \mathbb{R}^k}{\operatorname{argmin}} \; ||\Phi\theta - (R^\mu + \gamma P^\mu \Phi\theta^*)||_\epsilon^2$$

Setting the gradient of the to 0, we get

$$\Phi^T \Xi(\Phi\theta^* - (R^\mu + \gamma P^\mu \Phi\theta^*)) = 0$$

where $\Xi = \text{diag}\,(\epsilon_1, ..., \epsilon_n)$.
This is in fact the orthogonality condition from random signals.
Equivalently we can write

$$C\theta^* = d$$

where

$$C = \Phi^T \Xi (I - \gamma P^\mu)\Phi, \quad d = \Phi^T \Xi R^\mu$$

Solution approaches:

1. **Matrix inversion (LSTD)**:

   $$\theta^* = C^{-1} d$$

   In order to evaluate C, d, calculate a simulation-based approximation $\hat{C}_k, \hat{d}_k \to C, d$ by the LLN, and then use $\hat{\theta}_k = \hat{C}_k^{-1} \hat{d}_k$ – this is the Least Squares Temporal Difference algorithm (LSTD). Note that the simulation must use the policy μ, such that (after

some burn-in time) the states are samples from the distribution ϵ. Recall that

$$d = \Phi^T \Xi R^\mu = \sum_{s=1}^{|S|} \epsilon_s \phi(s) R^\mu(s).$$

So the following estimate converges to d:

$$\hat{d}_n = \frac{1}{n} \sum_{t=1}^{n} \phi(s_t) r(s_t, \mu(s_t)) =$$
$$= \frac{1}{n} \sum_{t=1}^{n} \sum_{s=1}^{|S|} \mathbf{1}(s_t = s) \phi(s) R^\mu(s) \underset{n \to \infty}{\longrightarrow} d.$$

Similarly,

$$\hat{C}_n = \frac{1}{n} \sum_{t=1}^{n} \phi(s_t)(I - \gamma P^\mu) \phi^T(s_t) \approx$$
$$\frac{1}{n} \sum_{t=1}^{n} \phi(s_t)(\phi^T(s_t) - \gamma \phi^T(s_{t+1})) \underset{n \to \infty}{\longrightarrow} C.$$

2. **Projected value iteration:**

$$\Phi \theta_{n+1} = \Pi_\epsilon T^\mu \Phi \theta_n = \Pi_\epsilon (R^\mu + \gamma P^\mu \Phi \theta_n),$$

which converges to θ^* since $\Pi_\epsilon T^\mu$ is a contraction operator.

The projection step is

$$\theta_{n+1} = \underset{\theta}{\operatorname{argmin}} ||\Phi \theta - (R^\mu + \gamma P^\mu \Phi \theta_n)||_\epsilon^2.$$

Setting the gradient to 0 w.r.t. θ:

$$\Phi^T \Xi (\Phi \theta_{n+1} - (R^\mu + \gamma P^\mu \Phi \theta_n)) = 0$$
$$\Rightarrow \theta_{n+1} = \theta_n - (\Phi^T \Xi \Phi)^{-1} (C \theta_n - d).$$

So we can approximate

$$\theta_{n+1} = \theta_n - G_n (C_n \theta_n - d_n),$$

where

$$G_n^{-1} \approx \frac{1}{n+1} \sum_{t=0}^{n} \phi(s_t) \phi(s_t)^T \Rightarrow G_n \approx (\Phi^T \Xi \Phi)^{-1}.$$

We observe that C_n, d_n, G_n can also be computed via the SA algorithm.

More generally, for the non-linear case, we have the iterative algorithm:

$$\hat{V}(\theta_{n+1}) = \Pi T^{\mu} \hat{V}(\theta_n),$$

where the projection Π here denotes a non-linear least squares fit, or even a non-parametric regression such as K-nearest neighbors. Convergence in this case is not guaranteed.

3. **Stochastic approximation – TD(0):** Consider the function-approximation variant of the TD(0) algorithm (cf. Section 6.4.1)

$$\theta_{n+1} = \theta_n + \alpha_n \underbrace{\left(r(s_n, \mu(s_n)) + \phi(s_{n+1})^\top \theta_n - \phi(s_n)^\top \theta_n\right)}_{\text{temporal difference}} \phi(s_n),$$

where the temporal difference term is the approximation (w.r.t. the weights at time n) of $r(s_n, \mu(s_n)) + V(s_{n+1}) - V(s_n)$.
This algorithm can be written as a stochastic approximation:

$$\theta_{n+1} = \theta_n + \alpha_n (C\theta_n - d + m_n),$$

where m_t is a noise term, and the corresponding ODE is $\dot{\theta} = C\theta - d$, with a unique stable fixed point at $\theta^* = C^{-1}d$. More details will be given in the tutorial.

Remark 9.4. *For a general (not necessarily linear) function approximation, the TD(0) algorithm takes the form:*

$$\theta_{n+1} = \theta_n + \alpha_n \left(r(s_n, \mu(s_n)) + f(s_{n+1}, \theta_n) - f(s_n, \theta_n)\right) \nabla_\theta f(s, \theta).$$

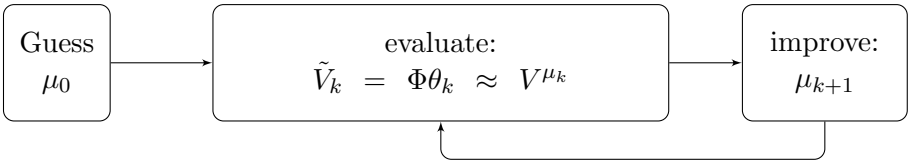
It can be derived as a stochastic gradient descent algorithm for the loss function

$$Loss(\theta) = ||f(s, \theta) - V^{\mu}(s)||_{\epsilon},$$

and replacing the unknown $V^{\mu}(s)$ with a Bellman-type estimator $r(s, \mu(s)) + f(s', \theta)$.

9.3.4 Approximate Policy Iteration

The algorithm: iterate between projection of V^{μ_k} onto $\mathcal{S}$ and policy improvement via a greedy policy update w.r.t. the projected V^{μ_k}.



The key question in approximate policy iteration, is how errors in the value-function approximation, and possibly also errors in the greedy policy update, affect the error in the final policy. The next result shows that if we can guarantee that the value-function approximation error is bounded at each step of the algorithm, then the error in the final policy will also be bounded. This result suggests that approximate policy iteration is a fundamentally sound idea.

Theorem 9.1. *If for each iteration k the policies are approximated well over $\mathcal{S}$:*

$$\max_s |\tilde{V}(s;\theta_k) - V^{\mu_k}(s)| \leq \delta,$$

and policy improvement approximates well

$$\max_s |T^{\mu_{k+1}}\tilde{V}(s;\theta_k) - T\tilde{V}(s;\theta_k)| < \epsilon,$$

Then

$$\limsup_k \max_s |V^{\mu_k}(s) - V^*(s)| \leq \frac{\epsilon + 2\gamma\delta}{(1-\gamma)^2}.$$

Online - SARSA

As we have seen earlier, it is easier to define a policy improvement step using the Q function. We can easily modify the TD(0) algorithm above to learn $\hat{Q}^\mu(s,a) = f(s,a;\theta)$:

$$\theta_{n+1} = \theta_n + \alpha_n \left(r(s_n,a_n) + f(s_{n+1},a_{n+1};\theta_n) - f(s_n,a_n;\theta_n)\right) \nabla_\theta f(s,a,\theta).$$

The actions are typically selected according to an $\epsilon-$greedy or softmax rule. Thus, policy evaluation is interleaved with policy improvement.

Batch - Least Squares Policy Iteration (LSPI)

One can also derive an approximate PI algorithm that works on a batch of data. Consider the linear case $\hat{Q}^\mu(s,a) = \theta^T\phi(s,a)$. The idea is to use LSTD(0) to iteratively fit $\hat{Q}^{\mu_k}$, where μ_k is the greedy policy w.r.t. $\hat{Q}^{\mu_{k-1}}$.

$$\hat{d}_n^k = \frac{1}{n}\sum_{t=1}^n \phi(s_t,a_t) r(s_t,a_t)$$

$$\hat{C}_n^k = \frac{1}{n}\sum_{t=1}^n \phi(s_t,a_t)(\phi^T(s_t,a_t) - \gamma\phi^T(s_{t+1},a_{t+1}^*)),$$

$$\theta_k = (\hat{C}_n^k)^{-1}\hat{d}_n^k.$$

where $a_{t+1}^* = \arg\max_a \hat{Q}^{\mu_{k-1}}(s_{t+1},a) = \arg\max_a \theta_{k-1}^T\phi(s_{t+1},a)$.

9.3.5 Approximate Value Iteration

Approximate VI algorithms directly approximate the optimal value function (or optimal Q function). Let us first consider the linear case. The idea in approximate VI is similar to the PBE, but replacing T^μ with T. That is, we seek solutions to the following projected equation:

$$\Phi\theta = \Pi T\{\Phi\theta\},$$

where Π is some projection. Recall that T is a contraction in the $\|.\|_\infty$ norm. Unfortunately, Π is not necessarily a contraction in $\|.\|_\infty$ for general function approximation.[1] Similarly, T is not a contraction in the $\|.\|_\epsilon$ norm. Thus, we have no guarantee that the projected equation has a solution. Nevertheless, algorithms based on this approach have achieved impressive success in practice.

For the non-linear case, we have:

$$\hat{Q}(\theta_{n+1}) = \Pi T\hat{Q}(\theta_n).$$

Online - Q learning

The function approximation version of online Q-learning resembles SARSA, only with an additional maximization over the next action:

$$\theta_{n+1} = \theta_n + \alpha_n \left(r(s_n, a_n) + \max_a f(s_{n+1}, a; \theta_n) - f(s_n, a_n; \theta_n)\right) \nabla_\theta f(s, a, \theta).$$

The actions are typically selected according to an $\epsilon-$greedy or softmax rule, to balance exploration and exploitation.

Batch – Fitted Q

In this approach, we iteratively project (fit) the Q function based on the projected equation:

$$\hat{Q}(\theta_{n+1}) = \Pi T\hat{Q}(\theta_n).$$

Assume we have a data set of samples $\{s_i, a_i, s'_i, r_i\}$,obtained from some data colletion policy. Then, the right hand side of the equation denotes a regression problem where the samples are: $X = \{s_i, a_i\}$ and $Y = \{r_i + \gamma \max_a \hat{Q}(s'_i, a; \theta_n)\}$. Thus, by solving a sequence of regression problems we approximate a solution to the projected equation.

Note that approximate VI algorithms are *off-policy* algorithms. Thus, in both Q-learning and fitted-Q, the policy that explores the MDP can be arbitrary (assuming of course it explores 'enough' interesting states).

[1] A restricted class of function approximators for which contraction does hold is called averagers, as was proposed in [3].

9.4 Exercises

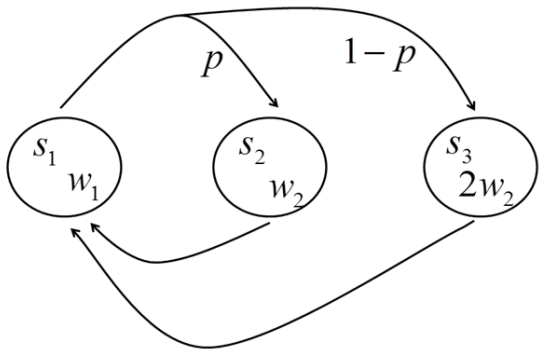
Exercise 9.1 (**Computational Complexity**)**.** What is the computational complexity (per-iteration, in terms of k – the number of features) of LSTD and TD(0)? Suggest an improvement to the matrix-inversion step of LSTD to reduce the computational complexity (hint: use the Sherman Morrison formula).

Exercise 9.2 (**Projected Bellman Operator**)**.** Consider the projection operator Π that projects a vector $V \in \mathbb{R}^n$ to a linear subspace S that is the span of k features $\phi_1(s), \ldots, \phi_k(s)$ w.r.t. the ϵ-weighted Euclidean norm. Also recall the Bellman operator for a fixed policy $T^\pi(v) \doteq r + \gamma P^\pi v$.

1. Show that for a vector $v \in S$, the vector $v' = T^\pi v$ is not necessarily in S. Choose an appropriate MDP and features to show this.

In class we have seen that ΠT^π is a contraction w.r.t. the ϵ-weighted Euclidean norm, when ϵ is the stationary distribution of P^π. We will now show that when ϵ is chosen differently, ΠT^π is not necessarily a contraction.

Consider the following 3-state MDP with zero rewards:



We consider a value function approximation $\tilde{V}(s) = \phi_1(s)w_1 + \phi_2(s)w_2$, given explicitly as $\tilde{V} = (w_1, w_2, 2w_2)^\top$, and we let $w = (w_1, w_2)^\top$ denote the weight vector.

2. What are the features $\phi(s)$ in this representation?

3. Write down the Bellman operator T^π explicitly. Write down$T^\pi \tilde{V}$.

4. What is the stationary distribution?

5. Write down the projection operator Π explicitly, for $\epsilon = \left(\frac{1}{2}, \frac{1}{4}, \frac{1}{4}\right)$.

6. Write an explicit expression for $\tilde{V}' \doteq \Pi T^{\pi} \tilde{V}$ in terms of w: the weight vector of $\tilde{V}$. Let $w' = (w_1', w_2')^{\top}$ denote the weights of $\tilde{V}'$. Write w' as a function of w .

7. Show that iteratively applying $\Pi T^{\pi}$to $\tilde{V}$ may diverge for certain values of p.

Exercise 9.3 (**SARSA with function approximation**)**.** In this question we will implement a reinforcement learning algorithm in a continuous domain using function approximation.

Recall the tabular SARSA algorithm (Section 6.4.5). We now present an extension of SARSA to the function approximation setting. Assume that we are given a set of state-action features $\phi(s, a) \in \mathbb{R}^k$. We propose to approximate $Q^{\pi}(s, a)$ as a linear combination of these features:

$$\tilde{Q}^{\pi}(s, a) = \sum_{i=1}^{k} \phi_i(s, a) w_i \equiv \phi(s, a)^{\top} w.$$

Our goal is to find the weight vector $w \in \mathbb{R}^k$.

In the SARSA algorithm, at each stage we observe $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$, simulated on-policy (i.e., by simulating the policy π) and update w by

$$\begin{aligned} w &:= w + \beta_t \delta_t \phi(s_t, a_t) \\ \delta_t &\stackrel{\Delta}{=} r(s_t) + \gamma \phi(s_{t+1}, a_{t+1})^{\top} w - \phi(s_t, a_t)^{\top} w \end{aligned}$$

where β_t is a suitable step-size, and γ is the discount factor.

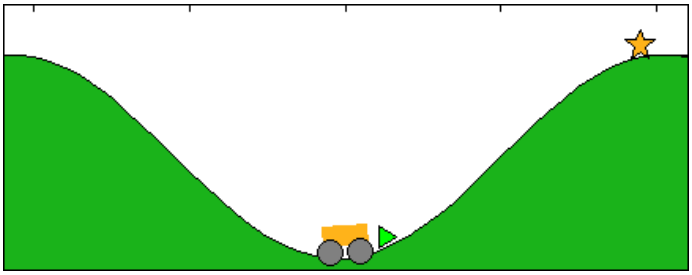
1. Explain the intuition for this update. You may use the TD(0) algorithm learned in class.

Policy improvement in SARSA is achieved by choosing the policy π as the ϵ-greedy policy with respect to the current w, that is, at time t the state is x_t and the action a_t is selected according to

$$a_t = \begin{cases} random & w.p.\ \epsilon \\ \arg\max_a \left(\phi(s_t, a)^{\top} w\right) & w.p.\ 1 - \epsilon \end{cases}.$$

2. Explain why the SARSA algorithm is expected to gradually improve performance.

We will now implement SARSA on a popular RL benchmark - *the mountain car*.



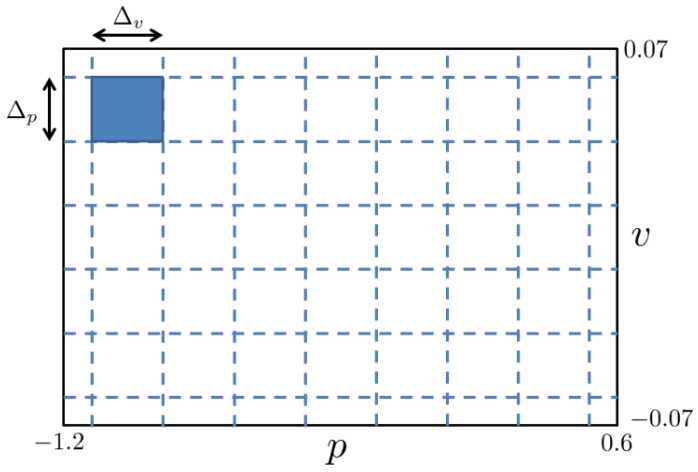
In the Mountain Car problem (see figure), an under-powered car must drive up a steep hill. Since gravity is stronger than the car's engine, even at full throttle, the car cannot simply accelerate up the steep slope. The car is situated in a valley and must learn to leverage potential energy by first driving up the left hill before gaining enough momentum to make it to the goal at the top of the right hill.

The state space is two-dimensional, and consists of the position $p \in [-1.2, 0.6]$ and velocity $v \in [-0.07, 0.07]$. There are 3 actions: $a = -1$ (accelerate left), $a = 0$ (don't accelerate), and $a = 1$ (accelerate right).

The simulation runs in episodes. At the beginning of an episode the initial state is $p_0 \sim Uniform\,[-0.5, 0.2]$, $v_0 \sim Uniform\,[-0.02, 0.02]$. If the car reaches the right hilltop: $p > 0.5$, the episode ends, and a reward $r = 5$ is received. At every other step the reward is $r = -1$. The maximum episode length is 500 steps.

The Matlab function `mountainCarSim(p, v, u)` (available at the course webpage) takes the current state and action and returns the next state of the car.

As function approximation, we will use grid-tiles. For each action a, we discretize the state space into n non-overlapping tiles of size $\Delta_p = 0.1, \Delta_v = 0.01$ (see figure), and we label the tiles $\psi_1^a, \ldots, \psi_n^a$.



Define binary **state-features** $\phi(s) \in \mathbb{R}^n$ by:

$$\phi_i(s) = \begin{cases} 1 & s \in \psi_i \\ 0 & else \end{cases} ,$$

and binary **state-action-features** $\phi(s,a) \in \mathbb{R}^k$ where $k = 3n$, by:

$$\begin{array}{l} \phi(s,-1) = \{\phi(s), 2n \text{ zeros}\} \\ \phi(s,0) = \{n \text{ zeros}, \phi(s), n \text{ zeros}\} \\ \phi(s,1) = \{2n \text{ zeros}, \phi(s)\} \end{array} .$$

3. Run the SARSA algorithm in this domain. Generate the following plots, and write a proper explanation for each plot.
   (a) Total reward (in episode) vs. episode
   (b) Goal was reached / not reached vs. episode
   (c) L_2 Norm of the weights w vs. episode
   (d) Trajectory of car (p vs. time) for the greedy policy starting from $(0,0)$, every 100 episodes of learning.
   (e) Final value function and policy after learning.

   You may use the following learning parameters for the algorithm:

- Step size: $\beta_t = \frac{100}{1000+\text{episodecount}}$
- Exploration: $\epsilon = 0.1$
- Discount: $\gamma = 0.95$
- Total number of learning episodes: $500 - 1000$
- Initial weights: zeros.

Bonus: Improve the learning algorithm using either:

- Different features - you may try:
  - Overlapping tiles (sometime called CMACs) (`http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node88.html#SECTION04232000000000000000`)
  - Radial Basis Functions (`http://en.wikipedia.org/wiki/Radial_basis_function`)
  - Fourier / polynomials (`http://irl.cs.duke.edu/fb.php`)
- Different algorithm - you may try:

  - SARSA(λ) e.g., `http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node89.html`
  - Q(λ) e.g., `http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node89.html`
- Any other idea that you like

Evidence that your method improves performance in some sense (learning rate/ final performance/ robustness to parameter changes/ etc.) will be rewarded with bonus points.

Chapter 10

Policy Gradient Methods

Recall that Reinforcement Learning is concerned with learning optimal control policies by interacting with the environment. So far, we have focused on learning methods that rely on learning the value function, from which the optimal policy can be inferred. Policy gradient methods take a different approach. Here a set of **policies** is directly parameterized by a continuous set of parameters, which are then optimized online. The basic algorithms use online gradient descent, hence achieve only local optimality in the parameter space. This also means that the set of possible policies is restricted, and might require a fair amount of prior knowledge for its effective definition. On the other hand, the restriction to a given set of policies simplifies the treatment of various aspects of the RL problem such as large or continuous state and action spaces. For these reasons, policy gradient methods are finding many applications in the area of robot learning. Policy Gradient algorithms belong to a larger class of Policy Search algorithms. An extensive survey can be found in:

- M. Deisenroth, G. Neumann and J. Peters, "A Survey on Policy Search for Robotics," Foundations and Trends in Robotics, Vol. 2, 2011, pp. 1-142.

10.1 Problem Description

We consider the standard MDP model in discrete time, with states $x_t \in X$, actions $u_t \in U$, transition kernel $\{p(x'|x,u)\}$, rewards $R_t = r_t(x_t, u_t)$, and policies $\pi \in \Pi$.

For concreteness, we consider an episodic (finite horizon) problem with the total return criterion, which is to be maximized:

$$J^\pi = \mathbb{E}^{\pi,x_0}(\sum_{t=0}^{T} R_t).$$

Here x_0 is a given initial state. We may allow the final time T to depend on the state (as in the Stochastic Shortest Path problem), as long as it is bounded.

We next assume that we are given parameterized set of policies: $\{\pi_\theta,\, \theta \in \Theta\}$, where θ is an I-dimensional parameter vector. We also refer to this set as a policy representation. The representation is often application-specific and should take into account prior knowledge on the problem.

We assume the following:

- The actions set U is continuous - typically a subset of $\mathbb{R}^l$. If the original action set is finite, we extend it to a continuous set by considering random (mixed) actions.
- The parameter set Θ is a continuous subset of $\mathbb{R}^I$.
- The policy set $\{\pi_\theta,\, \theta \in \Theta\}$ is smooth in θ. In particular, assuming each π_θ is a stationary policy, $\pi_\theta(s)$ is a continuous function of θ for each state s.

Policy representation: Some common generic representations include:

- Linear policies:
$$u = \pi_\theta(x) = \theta^T \phi(x),$$
where $\phi(x)$ is a suitable set of basis functions (or feature vectors).
- Radial Basis Function Networks:
$$\pi_\theta(x) = w^T \phi(x), \text{ with } \phi_i(x) = \exp(-\tfrac{1}{2}(x-\mu_i)^T D_i (x-\mu_i)),$$
where D_i is typically a diagonal matrix. The vector θ of tunable parameters includes w (the linear parameters), and possibly (μ_i) and (D_i) (nonlinear parameters).
- Neural Networks. For example, a Multi-Layer Perceptron (MLP):
$$\pi_\theta(x) = \psi(b_2 + w_2^T \psi(b_1 + w_1^T \phi(x))),$$
where ψ is a non-linear activation function (e.g., tanh or sigmoid) and the vector θ includes the weight and the bias terms (w_1, w_2, b_1, b_2) in the neural network. This can be seen as a generalization of the linear policy to include non-linearities.
- Logistic (a.k.a. softmax) functions: For a discrete (finite) action space, one can use the Boltzman-like probabilities
$$\pi_\theta(u|x) = \exp(w_u^T \phi(x)) / \sum\nolimits_{u'} \exp(w_{u'}^T \phi(x)),$$
where $\theta = (w_u)_{u \in U}$. It is convenient to designate one of the actions as 'anchor', with $w_u = 0$.

- Stochastic policies: in certain cases a stochastic policy is required for sufficient exploration of the system. This is typically modelled by adding noise to a deterministic policy. For example, in a linear policy with Gaussian noise:

$$P(u|x) = \pi_\theta(u|x) \sim \mathcal{N}\left(w^T \phi(x), \sigma\right),$$

  where the parameter vector θ contains w and the covariance matrix σ.

In robotics, a parameterized path is often described through a parameterized dynamical system, the output of which is used as a reference input to a feedback controller.

Gradient Updates: Plugging in the parameter-dependent policy π_θ in J^π, we obtain the parameter-dependent return function:

$$J(\theta) = J^{\pi_\theta}.$$

We wish to find a parameter θ that maximizes $J(\theta)$, at least locally. We discuss two high-level approaches for this task.

10.2 Search in Parameter Space

In this approach, the problem is viewed as a black-box optimization of the function $J(\theta)$. Black-box optimization (a.k.a. derivative free optimization) refers to optimization of an objective function through a black-box interface: the algorithm may only query the function $J(\theta)$ for a point θ; gradient information or the explicit form of $J(\theta)$ are not known.

In our setting, for each given value of the parameter vector θ, we can simulate or operate the system with control policy π_θ and measure the return $\hat{J}(\theta) = \sum_{t=0}^{T} R_t$, which gives a estimate of the expected return $J(\theta)$. We note that for stochastic systems or policies, this estimate will be noisy.

Note: The search in parameter space approach ignores the structure of our problem, namely, that trajectories are the result of rolling out a policy in an MDP. Only the mapping between θ and the resulting cumulative reward is considered.

Many black-box optimization algorithms have been proposed in the literature. We describe several that have been popular in RL literature.

10.2.1 Gradient Approximation

These methods use function evaluations of $J(\theta)$ to approximate the gradient $\nabla_\theta J(\theta)$. Given the gradient, we may consider a gradient ascent scheme, of the form

$$\theta_{k+1} = \theta_k + \alpha_k \nabla_\theta J(\theta_k). \tag{10.1}$$

Here (α_k) is the gain (or learning rate) sequence, and

$$\nabla_\theta J(\theta) = \frac{\partial J(\theta)}{\partial \theta}$$

is the gradient of the return function with respect to θ.

It remains of course to determine how to compute the required gradient. We next outline two options.

Note: Whenever required, we assume without further mention that $J(\theta)$ is continuously differentiable, and that the expectation and derivative in its definition can be interchanged.

Finite Difference Methods

Finite difference methods are among the most common and straightforward methods for estimating the gradient in a variety of applications.

Suppose that, for each given value of the parameter vector θ, we can simulate or operate the system with control policy π_θ and measure the return $\hat{J}(\theta) = \sum_{t=0}^{T} R_t$, which gives a estimate of the expected return $J(\theta)$. We note that this estimate will be noisy when either the policy or the system contain random moves.

Component-wise gradient estimates: We can now obtain a noisy estimate of each component of the gradient vector using a finite difference of the form:

$$\frac{\partial J(\theta)}{\partial \theta_i} \approx \frac{\hat{J}(\theta + \delta e_i) - \hat{J}(\theta)}{\delta},$$

or, preferably, the symmetric difference

$$\frac{\partial J(\theta)}{\partial \theta_i} \approx \frac{\hat{J}(\theta + \delta e_i) - \hat{J}(\theta - \delta e_i)}{2\delta}.$$

Note:

- Since the estimates $\hat{J}$ are typically noisy, repeated trials and averaging of several such differences are needed to reduce the estimation error. However, one can also use the noisy estimates with a low gain in the gradient ascent scheme, which has a similar averaging effect.
- The choice of the step size δ is crucial, as it controls the tradeoff between the relative noise level and the non-linearity of $J(\theta)$. We will not get into this issue here.
- A useful method to reduce the variance of the above difference is to use coupled random simulation, meaning that the random variables that govern the random choices in the simulation (or action choices) are drawn only once, and used for both estimates $\hat{J}(\theta + \delta e_i)$ and $\hat{J}(\theta - \delta e_i)$. These and other variance reduction methods are standard tools in the area of Monte-Carlo simulation.

Least-squared gradient estimates: Suppose we simulates/operate the system with a set of parameters $\theta^{[k]} = \theta + \Delta\theta^{[k]}$, $1 \le k \le K$, to obtain a corresponding set of reward estimates $\hat{J}^{[k]} = \hat{J}(\theta + \Delta\theta^{[k]})$.

We can now use the linear relations

$$\hat{J}(\theta + \Delta\theta^{[k]}) \approx J(\theta) + \Delta\theta^{[k]} \cdot \nabla J(\theta) \tag{10.2}$$

to obtain a least-squares estimate of $\nabla J(\theta)$. For example, if an estimate $\hat{J}(\theta)$ is pre-computed, the relation

$$\Delta J^{[k]} \stackrel{\Delta}{=} \hat{J}(\theta + \Delta\theta^{[k]}) - \hat{J}(\theta) \approx \Delta\theta^{[k]} \cdot \nabla J(\theta)$$

leads to the LS estimate

$$\hat{\nabla} J(\theta) = (\mathbf{\Delta\Theta}^T \mathbf{\Delta\Theta})^{-1} \mathbf{\Delta\Theta}^T \mathbf{\Delta J},$$

where $\mathbf{\Delta\Theta} = [\Delta\theta^{[1]}, \ldots, \Delta\theta^{[K]}]^T$, and $\mathbf{\Delta J} = [\Delta J^{[1]}, \ldots, \Delta J^{[K]}]^T$. (Note that we consider here $\Delta\theta^{[k]}$ as a column vector, so that $\mathbf{\Delta\Theta}$ is a $K \times I$ matrix).

If $\hat{J}(\theta)$ is not given in advance, we can use directly the relations from (10.2) in matrix form,

$$\hat{\mathbf{J}} \stackrel{\Delta}{=} \begin{bmatrix} \hat{J}^{[1]} \\ \vdots \\ \hat{J}^{[K]} \end{bmatrix} \approx M \begin{bmatrix} J(\theta) \\ \nabla J(\theta) \end{bmatrix}, \qquad M = [\mathbf{1}, \mathbf{\Delta\Theta}]$$

to obtain the joint estimate $(M^T M)^{-1} M \hat{\mathbf{J}}$ for $J(\theta)$ and $\nabla J(\theta)$.

10.3 Population Based Methods

This family of methods maintain a distribution over the parameter vector θ, and use function evaluations to 'narrow in' the distribution on an optimal choice. Assume that we have a parametrized distribution over the policy parameters $P_\phi(\theta)$. Note the distinction between the policy parameters θ and the parameters ϕ for the distribution of θ values.

For example, $P_\phi(\theta)$ can be a multivariate Gaussian, where ϕ encodes the mean and covariance of the Gaussian.

Cross Entropy Method (CEM) This method updates ϕ iteratively according to the following scheme:

1. Given current parameter ϕ_i, sample N population members

$$\theta_k \sim P_{\phi_i}(\theta), \quad k = 1, \ldots, N.$$

2. For each k, simulate the system with π_{θ_k} and measure the return $J(\theta_k)$.

3. Let K^* denote a set of the $p\%$ top performing parameters

4. Improve the parameter:

$$\phi_{i+1} = \arg\max_{\phi} \sum_{k \in K^*} \log P_\phi(\theta_k). \qquad (\star)$$

The intuition here is that by refitting the distribution P_ϕ to the top performing parameters in the population, we are iteratively improving the distribution.

Note that other forms of the improvement step $(\star)$ have been proposed in the literature. For example, in Reward Weighted Regression (RWR) the parameters are updated according to

$$\phi_{i+1} = \arg\max_{\phi} \sum_{k \in 1,\ldots,N} J(\theta_k) \log P_\phi(\theta_k).$$

10.4 Likelihood Ratio Methods

Likelihood ratio-based methods allow to obtain a (noisy) estimate of the reward gradient from a **single** trial. The approach is again standard in the Monte-Carlo simulation area, where it is also known as the Score Function methods. Its first use in the controlled process (or RL) context is known as the REINFORCE algorithm. Interestingly, the RL formulation of this method can exploit the MDP structure of the problem, by using dynamic programming ideas to reduce variance in the estimation.

Let $\tau = (x_0, u_0, \ldots, x_T)$ denote the process history, or sample path, of a single run of our episodic problem. For simplicity we consider here a discrete model (finite state and action sets). Let $p_\theta(\tau)$ denote the probability mass function induced on the process by the policy π_θ. That is, assuming π_θ is a Markov policy,

$$p_\theta(\tau) = p(x_0) \prod_{t=0}^{T-1} \pi_\theta(u_t|x_t) p(x_{t+1}|x_t, u_t).$$

Denote $R(\tau) = \sum_{t=0}^{T-1} r(x_t, u_t) + r_T(x_T)$, so that

$$J(\theta) = \mathbb{E}^\theta(R(\tau)) = \sum\nolimits_\tau R(\tau) p_\theta(\tau)\,.$$

Observe now that$\nabla_\theta p_\theta(\tau) = p_\theta(\tau)\, \nabla_\theta \log p_\theta(\tau)$. Therefore, assuming that the expectation and derivative can be interchanged,

$$\begin{aligned} \nabla J(\theta) &= \sum\nolimits_\tau R(\tau) \nabla_\theta p_\theta(\tau) \\ &= \sum\nolimits_\tau \left[R(\tau) \nabla_\theta \log p_\theta(\tau)\right] p_\theta(\tau) \\ &= \mathbb{E}^\theta \left(R(\tau) \nabla_\theta \log p_\theta(\tau)\right). \end{aligned}$$

Furthermore, observing the above expression for $p_\theta(\tau)$,

$$\nabla_\theta \log p_\theta(\tau) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(u_t|x_t).$$

Importantly, the latter expression depends only on the derivative of the control policy, which is known to us, and not on the (unknown) process dynamics and reward function.

We can now obtain an unbiased estimate of $\nabla J(\theta)$ as follows:

- Simulate/implement a single episode ("rollout") of the controlled system with policy π_θ.
- Compute $R(\tau)$ as $R(\tau) = \sum_{t=0}^{T-1} r(x_t, u_t) + r_T(x_T)$, or directly using the observed rewards $R(\tau) \triangleq \sum_{t=0}^{T} R_t$.
- Compute $\hat{\nabla} J(\theta) = R(\tau) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(u_t|x_t)$ This is typically a noisy estimate, which can of course be improved by averaging over repeated trials.

Illustrative Example

Consider the bandit setting, where the MDP has only a single state and the horizon is $T = 1$. The policy and reward are given as follows:

$$\begin{aligned} r(u) &= u, \\ \pi_\theta(u) &= \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(u-\theta)^2}{2\sigma^2}). \end{aligned}$$

We have that $J(\theta) = \mathbb{E}[u] = \theta$, and thus $\nabla_\theta J(\theta) = 1$. Using the policy gradient formula, we calculate:

$$\begin{aligned} \nabla_\theta \log \pi_\theta(u) &= \frac{u-\theta}{\sigma^2}, \\ \nabla_\theta J(\theta) &= \mathbb{E}[\frac{u(u-\theta)}{\sigma^2}] \\ &= \frac{1}{\sigma^2}(\mathbb{E}[u^2] - (\mathbb{E}[u])^2) = 1. \end{aligned}$$

Note the intuitive interpretation of the policy gradient here: we average the change to the mean action $u - \theta$ and the reward it produces $r(u) = u$. In this case, actions above the mean lead to higher reward, thereby 'pushing' the mean action θ to increase. Also note the relation to the reward-weighted regression expression above.

10.4.1 Variance Reduction

We now provide a generalization of the policy gradient. We will consider an episodic MDP setting, and assume that for every policy parameter θ, an absorbing state is reached w.p. 1. We can therefore replace the finite horizon return with an infinite sum $J^\pi = \mathbb{E}^{\pi,x_0}(\sum_{t=0}^{\infty} R_t)$. We also recall the value and action value functions $V^\pi(x)$ and $Q^\pi(x,u)$.

Proposition 10.1. *The policy gradient can be written as:*

$$\nabla_\theta J(\theta) = \mathbb{E}^\theta \left(\sum_{t=0}^{\infty} \Psi_t \nabla_\theta \log \pi_\theta(u_t|x_t) \right),$$

where Ψ_t can be either one of the following terms:

1. *Total reward, $\sum_{t=0}^{\infty} r(x_t,u_t) - b(x_t)$, where b is a state-dependent baseline*
2. *Future reward following action u_t, $\sum_{t'=t}^{\infty} r(x_t,u_t) - b(x_t)$*
3. *State-action value function, $Q^\pi(x_t,u_t)$*
4. *Advantage function, $Q^\pi(x_t,u_t) - V^\pi(x_t)$*
5. *Temporal difference, $r(x_t,u_t) + V^\pi(x_{t+1}) - V^\pi(x_t)$*

All the above formulations are unbiased estimates of the policy gradient. However, they differ in their variance, and variance reduction plays an important role in practical applications[1]. Let illustrate this in the previous bandit example.

Illustrative Example (cont'd)

Consider the previous bandit setting, where we recall that $r(u) = u, \quad \pi_\theta(u) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(u-\theta)^2}{2\sigma^2})$. Find a fixed baseline b that minimizes the variance of the policy gradient estimate.

The policy gradient formula in this case is:

$$\nabla_\theta J(\theta) = \mathbb{E}[\frac{(u-b)(u-\theta)}{\sigma^2}] = 1,$$

and we can calculate the variance

$$\begin{aligned}
\frac{1}{\sigma^4}\mathrm{Var}\left[(u-b)(u-\theta)\right] &= \frac{1}{\sigma^4}\mathbb{E}\left[\left((u-b)(u-\theta)\right)^2 - 1\right] \\
&= \frac{1}{\sigma^4}\mathbb{E}\left[\left((u-\theta)(u-\theta) + (\theta-b)(u-\theta)\right)^2 - 1\right] \\
&= \frac{1}{\sigma^4}\mathbb{E}\left[(u-\theta)^4 + (\theta-b)(u-\theta)^3 + (\theta-b)^2(u-\theta)^2 - 1\right] \\
&= \frac{1}{\sigma^4}\mathbb{E}\left[(u-\theta)^4 + (\theta-b)^2(u-\theta)^2 - 1\right],
\end{aligned}$$

[1] In the simulation literature, the technique of changing the variance of the estimate by adding terms that do not change its bias is known as *control variates*.

which is minimized for $b = \theta$.

Note 1: The average reward, state-action value function, and advantage function baselines are commonly used in practice. While in our illustrative example the average reward was optimal, in general it is not necessarily so. An in-depth discussion of this topic can be found in:

- Greensmith, E., Bartlett, P.L. and Baxter, J., 2004. Variance reduction techniques for gradient estimates in reinforcement learning. Journal of Machine Learning Research, 5(Nov), pp.1471-1530.

Nevertheless, these baselines often lead to significant performance gains in practice.

Note 2: Typically, the value functions V^π and Q^π will not be known, and will have to be estimated along with the policy gradient using TD methods or regression. An important question here is how the error in value function estimation affects the policy gradient bias. There exist special policy classes and function approximators that are said to be *compatible*, where the value estimation error is orthogonal to the policy gradient bias. In general, however, this is not the case.

Proof of Proposition 10.1

Proof. We will start by establishing a useful property. Let $p_\theta(z)$ be some parametrized distribution. Differentiating $\sum_z p_\theta(z) = 1$ yields

$$\sum\nolimits_z (\nabla \log p_\theta(z)) p_\theta(z) = \mathbb{E}^\theta(\nabla \log p_\theta(z)) = 0. \tag{10.3}$$

We can now observe that adding a state-dependent baseline to the policy gradient does not add bias:

$$\begin{aligned}
\mathbb{E}^\theta\left(\sum_{t=0}^{\infty} b(x_t)\nabla_\theta \log \pi_\theta(u_t|x_t)\right) &= \sum_{t=0}^{\infty} \mathbb{E}^\theta\left[b(x_t)\nabla_\theta \log \pi_\theta(u_t|x_t)\right] \\
&= \sum_{t=0}^{\infty} \mathbb{E}^\theta\left[\mathbb{E}^\theta\left[b(x_t)\nabla_\theta \log \pi_\theta(u_t|x_t)\middle| x_t\right]\right] \\
&= \sum_{t=0}^{\infty} \mathbb{E}^\theta\left[b(x_t)\mathbb{E}^\theta\left[\nabla_\theta \log \pi_\theta(u_t|x_t)\middle| x_t\right]\right] \\
&= 0,
\end{aligned}$$

where the last equation follows from applying (10.3) to the inner expectation. Note that the justification for exchanging the expectation and infinite sum in the first equality is not straightforward. In this case it can be shown to hold by the Fubini theorem, using the assumption that every trajectory reaches a terminal state in a bounded time w.p. 1.

We continue to show the independence on past rewards. We have that

$$
\begin{aligned}
&\mathbb{E}^\theta\left[\sum_{t=0}^{\infty}\sum_{t'=0}^{t-1} r(x_{t'},u_{t'})\nabla_\theta \log \pi_\theta(u_t|x_t)\right] \\
&= \sum_{t=0}^{\infty}\mathbb{E}^\theta\left[\sum_{t'=0}^{t-1} r(x_{t'},u_{t'})\nabla_\theta \log \pi_\theta(u_t|x_t)\right] \\
&= \sum_{t=0}^{\infty}\mathbb{E}^\theta\left[\mathbb{E}^\theta\left[\sum_{t'=0}^{t-1} r(x_{t'},u_{t'})\nabla_\theta \log \pi_\theta(u_t|x_t)\,\middle|\, x_0,u_0,\ldots,x_t\right]\right] \\
&= \sum_{t=0}^{\infty}\mathbb{E}^\theta\left[\sum_{t'=0}^{t-1} r(x_{t'},u_{t'})\mathbb{E}^\theta\left[\nabla_\theta \log \pi_\theta(u_t|x_t)|\, x_0,u_0,\ldots,x_t\right]\right] \\
&= \sum_{t=0}^{\infty}\mathbb{E}^\theta\left[\sum_{t'=0}^{t-1} r(x_{t'},u_{t'})\mathbb{E}^\theta\left[\nabla_\theta \log \pi_\theta(u_t|x_t)|\, x_t\right]\right] \\
&= 0,
\end{aligned}
$$

where in the second to last equality we used the Markov property, and in the last equality we again applied (10.3). We have thus proved (1) and (2).

Exercise 1: where would this derivation fail for future rewards?

We continue to prove (3).

$$
\begin{aligned}
&\mathbb{E}^\theta\left[\sum_{t=0}^{\infty}\sum_{t'=t}^{\infty} r(x_{t'},u_{t'})\nabla_\theta \log \pi_\theta(u_t|x_t)\right] \\
&\sum_{t=0}^{\infty}\mathbb{E}^\theta\left[\sum_{t'=t}^{\infty} r(x_{t'},u_{t'})\nabla_\theta \log \pi_\theta(u_t|x_t)\right] \\
&\sum_{t=0}^{\infty}\mathbb{E}^\theta\left[\mathbb{E}^\theta\left[\sum_{t'=t}^{\infty} r(x_{t'},u_{t'})\nabla_\theta \log \pi_\theta(u_t|x_t)\,\middle|\, x_t,u_t\right]\right] \\
&\sum_{t=0}^{\infty}\mathbb{E}^\theta\left[\nabla_\theta \log \pi_\theta(u_t|x_t)\mathbb{E}^\theta\left[\sum_{t'=t}^{\infty} r(x_{t'},u_{t'})\,\middle|\, x_t,u_t\right]\right] \\
&\sum_{t=0}^{\infty}\mathbb{E}^\theta\left[\nabla_\theta \log \pi_\theta(u_t|x_t)Q^\pi(x_t,u_t)\right].
\end{aligned}
$$

Exercise 2: Prove (4) and (5).

□

10.4.2 Natural Policy Gradient

In the gradient ascent scheme (10.1), the idea is to take small steps that iteratively improve the policy. The question is, what is the best metric to define 'small steps' in? Taking a step η in the gradient direction is equivalent to solving the following optimization problem:

$$\begin{aligned} \arg\max_{\Delta\theta} \quad & \Delta\theta^\top \nabla J(\theta), \\ s.t. \quad & \Delta\theta^\top \Delta\theta \leq \eta. \end{aligned} \tag{10.4}$$

Thus, standard gradient ascent takes a small improvement step w.r.t. a Euclidean distance in the parameter space. However, this scheme can be highly sensitive to the specific parametrization employed - it might be that a small change in parameters causes a very drastic change to the behavior of the policy. The natural gradient attempts to rectify this situation by replacing the Euclidean distance between two parameters θ and $\theta + \Delta\theta$ by the Kullback-Leibler distance[2] between the probability distributions $p_\theta(\tau)$ and $p_{\theta+\Delta\theta}(\tau)$ induced by these parameters. Using a Taylor expansion, the KL distance can be approximated as

$$D_{KL}(p_\theta(\tau)||p_{\theta+\Delta\theta}(\tau)) \approx \Delta\theta^\top F_\theta \Delta\theta,$$

where F_θ is the Fisher Information Matrix, $F_\theta = \sum_\tau p_\theta(\tau) \nabla \log p_\theta(\tau) \nabla \log p_\theta(\tau)^\top$.

Replacing the constraint in (10.4) with $\Delta\theta^\top F_\theta \Delta\theta \leq \eta$ leads to a modified gradient definition known as the Natural Gradient: :

$$\nabla^N J(\theta) = F_\theta^{-1} \nabla J(\theta).$$

Note that the Fisher Information Matrix can be calculated by sampling, since $\log p_\theta(\tau)$ only requires knowing the policy (as in the policy gradient derivation above). Natural policy gradient schemes lead in general to faster and more robust convergence to the optimal policy.

[2]The KullbackLeibler (KL) distance between two distributions P, Q is defined as $D_{KL}(P||Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)}$. It is a standard tool in information theory.

Bibliography

[1] M. Kearns, Y. Mansour, and A. Y. Ng, "A sparse sampling algorithm for near-optimal planning in large markov decision processes," *Machine learning*, vol. 49, no. 2-3, pp. 193–208, 2002.

[2] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning*, pp. 282–293, Springer, 2006.

[3] G. J. Gordon, "Stable function approximation in dynamic programming," in *Machine Learning Proceedings 1995*, pp. 261–268, Elsevier, 1995.