Optimization Algorithms

Optimization is a fundamental concept in mathematical modeling and computational science, focusing on finding the best solution from a set of feasible solutions. It is a crucial tool across various domains, including operations research, engineering, machine learning, and artificial intelligence. In the context of data science, optimization serves as a powerful framework for solving complex problems where efficiency, accuracy, and feasibility are critical considerations.

At its core, optimization involves defining an objective function—representing a quantity to be maximized or minimized—while satisfying a set of constraints that restrict the solution space. Optimization algorithms vary in complexity and application, ranging from simple linear programming problems to more sophisticated techniques such as quadratic programming (QP) and integer programming (IP). Many real-world problems, such as supply chain logistics, financial portfolio management, and resource allocation in healthcare, rely on optimization to achieve optimal outcomes.

Beyond standalone applications, optimization plays a pivotal role in the derivation of advanced modeling techniques. Support Vector Machines (SVMs), for example, leverage quadratic programming to find the optimal hyperplane for classification. Similarly, neural networks employ optimization algorithms like gradient descent to iteratively adjust model parameters and minimize loss functions. Dynamic programming, another key optimization technique, is fundamental in decision-making problems, including reinforcement learning and Markov Decision Processes (MDPs).

Contents

1 Mathematical Foundations **1**
- 1.1 Convex Sets and Convex Hulls . . . 1
- 1.2 Convex Functions . . . 1
- 1.3 Linear Algebra Concepts in Optimization . . . 1
- 1.4 Gradient and Hessian in Optimization . . . 2

2 Linear Programming **2**
- 2.1 Formulation of a Linear Program . . . 3
- 2.2 Graphical Solution for Two Variables . . . 3
- 2.3 The Simplex Algorithm . . . 4
- 2.4 Linear Programming in Python . . . 4

3 Integer Programming **4**
- 3.1 Formulation of Integer Programming Problems . . . 5
- 3.2 Methods for Solving Integer Programming Problems . . . 5
- 3.3 Example: Task Assignment Problem . . . 5
- 3.4 Integer Programming in Python . . . 6

4 Quadratic Programming **7**
- 4.1 Formulation of Quadratic Programming Problems . . . 7
- 4.2 Solution Methods for Quadratic Programming . . . 8
- 4.3 Example: Portfolio Optimization . . . 8
- 4.4 Quadratic Programming in Python . . . 8

5 Introduction to Dynamic Programming **9**

6 Markov Decision Processing **10**
- 6.1 Agent-Based Systems . . . 10
- 6.2 Definition of an MDP . . . 11
- 6.3 Determining the Value of a Policy . . . 12
- 6.4 Finding an Optimal Policy . . . 12
- 6.5 Solving MDPs . . . 12
- 6.6 Value Iteration . . . 13
- 6.7 Policy Iteration . . . 14
- 6.8 Convergence of Synchronous Dynamic Programming . . . 15

7 Heuristic and Metaheuristic Methods **15**
- 7.1 Heuristic Methods . . . 15
- 7.2 Metaheuristic Methods . . . 16
  - 7.2.1 Simulated Annealing (SA) . . . 16
  - 7.2.2 Genetic Algorithms (GA) . . . 16
  - 7.2.3 Particle Swarm Optimization (PSO) . . . 16
  - 7.2.4 Ant Colony Optimization (ACO) . . . 17
- 7.3 Comparison and Practical Use Cases . . . 17

8 Conclusion 17

1 Mathematical Foundations

Optimization relies heavily on mathematical principles, particularly in the areas of convexity, linear algebra, and differentiation. These concepts provide the theoretical backbone for formulating and solving optimization problems efficiently.

1.1 Convex Sets and Convex Hulls

A set S in $\mathbb{R}^n$ is convex if, for any two points $x_1, x_2 \in S$, the line segment connecting them is also contained within S. Mathematically, this is expressed as:

$$\lambda x_1 + (1-\lambda)x_2 \in S, \quad \forall \lambda \in [0,1]. \tag{1}$$

The convex hull of a set of points is the smallest convex set that contains all the given points. It plays an important role in approximating non-convex sets with convex boundaries to facilitate optimization.

1.2 Convex Functions

A function $f(x)$ is convex if its domain is a convex set and it satisfies the Jensen's inequality:

$$f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda) f(x_2), \quad \forall \lambda \in [0,1]. \tag{2}$$

Convex functions guarantee that any local minimum is also a global minimum, making optimization more tractable. Additional conditions for convexity include:

- **First-order condition:** A differentiable function $f(x)$ is convex if:

$$f(y) \geq f(x) + \nabla f(x)^T (y - x), \quad \forall x, y. \tag{3}$$

- **Second-order condition:** A twice-differentiable function $f(x)$ is convex if its Hessian matrix $H(x)$ is positive semi-definite:

$$H(x) \geq 0, \quad \text{where } H(x) = \nabla^2 f(x). \tag{4}$$

1.3 Linear Algebra Concepts in Optimization

Linear algebra is fundamental to optimization problems, particularly in linear and quadratic programming. Some key concepts include:

- **Matrix-vector multiplication:** The foundation for expressing systems of linear equations, often written as:

$$Ax = b, \tag{5}$$

  where A is an $m \times n$ matrix, x is an $n \times 1$ vector of variables, and b is an $m \times 1$ vector.

- **Dot product:** Measures similarity between two vectors:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i. \tag{6}$$

- **Norms:** Measure the length or size of a vector:

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}. \tag{7}$$

1.4 Gradient and Hessian in Optimization

The gradient and Hessian are critical tools for characterizing and solving optimization problems.

- **Gradient:** The gradient of a function $f(x)$, denoted $\nabla f(x)$, is a vector of partial derivatives representing the direction of steepest ascent:

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}. \tag{8}$$

- **Hessian:** The Hessian matrix $H(x)$ contains second-order partial derivatives and provides information about curvature:

$$H(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}. \tag{9}$$

  If $H(x)$ is positive definite, then $f(x)$ has a local minimum at x.

These mathematical foundations form the basis for understanding and implementing optimization algorithms in both theoretical and practical contexts.

2 Linear Programming

Linear programming (LP) serves as an excellent starting point for understanding optimization due to its structured yet simple formulation. It provides a clear framework for defining and solving optimization problems, making it an essential foundation before delving into more complex methods. At its core, an optimization problem consists of three fundamental components: decision variables, which represent the choices available; an objective function, which defines the criterion to be maximized or minimized; and constraints, which limit the set of feasible solutions based on problem-specific restrictions.

LP is particularly valuable because it allows for an intuitive grasp of these components within a linear structure, where both the objective function and constraints exhibit a straightforward mathematical form. This simplicity enables efficient computational solutions while still addressing a wide range of real-world applications, from resource allocation in economics to production planning in engineering and logistics.

Understanding LP is critical as it lays the groundwork for more advanced techniques such as integer programming, quadratic programming, and nonlinear optimization. Throughout this document, we will build on these foundational concepts, progressively introducing more sophisticated methods

that incorporate additional complexity while preserving the core principles established by LP. The simplicity of LP allows for a clear examination of the key components of an optimization problem, including decision variables, an objective function, and constraints that define the feasible region. By studying LP, we establish a foundation for more advanced techniques such as integer programming, quadratic programming, and nonlinear optimization, where complexity arises due to additional structural constraints or nonlinearity.

LP is widely applied across various domains, including economics, operations research, logistics, and engineering, demonstrating its versatility in modeling and solving problems involving resource allocation, scheduling, and cost minimization. As we progress through this document, we will build upon the principles of LP to explore more sophisticated optimization methods that accommodate more complex problem structures.

2.1 Formulation of a Linear Program

A standard linear programming problem is formulated as:

$$\max \text{ or } \min c^T x \tag{10}$$

subject to:

$$Ax \leq b, \quad x \geq 0, \tag{11}$$

where:

- $x \in \mathbb{R}^n$ is the vector of decision variables.
- $c \in \mathbb{R}^n$ is the cost (or profit) vector.
- $A \in \mathbb{R}^{m \times n}$ is the matrix of constraints.
- $b \in \mathbb{R}^m$ is the constraint vector.

2.2 Graphical Solution for Two Variables

For problems with two decision variables, a graphical approach can be used. The feasible region is identified by plotting constraint inequalities, and the optimal solution is found at one of the vertices of the feasible region.

Example:

Consider the following linear program:

$$\begin{aligned} \max \quad & z = 3x_1 + 5x_2 && (12)\\ \text{subject to:} \quad & x_1 + 2x_2 \leq 8 && (13)\\ & 3x_1 + 2x_2 \leq 12 && (14)\\ & x_1, x_2 \geq 0. && (15) \end{aligned}$$

Solution Steps:

1. Plot the constraint lines $x_1 + 2x_2 = 8$ and $3x_1 + 2x_2 = 12$ on a coordinate plane.
2. Identify the feasible region where all constraints are satisfied.

3. Locate the vertices of the feasible region and evaluate z at each vertex.
4. The vertex that maximizes z is the optimal solution.

For this example, solving the system at the feasible region's extreme points yields an optimal solution at $(x_1, x_2) = (2, 3)$, with an optimal objective function value $z = 3(2) + 5(3) = 21$.

2.3 The Simplex Algorithm

For larger problems, graphical methods are impractical. The simplex algorithm is a widely used iterative method that systematically moves along the edges of the feasible region to find the optimal solution.

The main steps of the simplex method are:

- Convert constraints into an augmented system with slack variables.
- Identify an initial basic feasible solution.
- Perform pivoting operations to improve the solution iteratively.
- Stop when no further improvement is possible.

The simplex method has polynomial-time complexity in the average case and is highly efficient for solving practical LP problems.

2.4 Linear Programming in Python

A practical implementation of LP can be done using Python's `scipy.optimize.linprog` function. Below is an example solving our earlier problem:

```
from scipy.optimize import linprog

c = [-3, -5]  # Coefficients for maximization (negated for linprog)
A = [[1, 2], [3, 2]]
b = [8, 12]

x_bounds = [(0, None), (0, None)]  # Non-negativity constraints

result = linprog(c, A_ub=A, b_ub=b, bounds=x_bounds, method='simplex')

print("Optimal Value:", -result.fun)  # Reversing negation for correct value
print("Optimal Solution:", result.x)
```

This code will return the optimal solution and objective value, demonstrating how LP problems can be efficiently solved using computational tools.

3 Integer Programming

Integer Programming (IP) extends the principles of Linear Programming (LP) by introducing additional constraints that require some or all decision variables to take integer values. This added

complexity significantly enhances its applicability to real-world problems but also increases computational difficulty. Integer constraints make IP problems inherently combinatorial, meaning that traditional LP solution techniques, such as the Simplex method, are no longer directly applicable.

3.1 Formulation of Integer Programming Problems

A general Integer Programming problem is formulated as:

$$\max \text{ or } \min c^T x \tag{16}$$

subject to:

$$Ax \leq b, \quad x \in \mathbb{Z}^n, \quad x \geq 0, \tag{17}$$

where:

- $x \in \mathbb{Z}^n$ denotes that all decision variables must take integer values.
- $c \in \mathbb{R}^n$ represents the objective function coefficients.
- $A \in \mathbb{R}^{m \times n}$ is the constraint matrix.
- $b \in \mathbb{R}^m$ is the constraint vector.

There are different types of Integer Programming:

- **Pure Integer Programming:** All decision variables are restricted to integer values.
- **Mixed Integer Programming (MIP):** Some variables can take continuous values while others remain integer.
- **Binary Integer Programming:** Variables are restricted to binary values (0 or 1), useful for decision-making problems.

3.2 Methods for Solving Integer Programming Problems

Since integer constraints impose a combinatorial nature to the solution space, specialized techniques are required to solve IP problems efficiently:

- **Branch and Bound:** A systematic enumeration method that partitions the feasible region into smaller subproblems and prunes infeasible solutions.
- **Cutting Plane Method:** Iteratively refines the feasible region by adding linear constraints (cuts) to exclude non-integer solutions from LP relaxations.
- **Heuristic and Metaheuristic Methods:** Approximations such as Genetic Algorithms, Simulated Annealing, and Particle Swarm Optimization provide near-optimal solutions for large-scale problems where exact methods become impractical.

3.3 Example: Task Assignment Problem

Consider a scenario where tasks must be assigned to workers while minimizing the total cost, subject to constraints that each task is assigned to exactly one worker and each worker is assigned a limited number of tasks.

Mathematical Formulation: Let x_{ij} be a binary decision variable representing whether task j is assigned to worker i ($x_{ij} \in \{0, 1\}$). The problem formulation is:

$$\min \quad \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{18}$$

$$\text{subject to:} \quad \sum_{i=1}^{m} x_{ij} = 1, \quad \forall j \quad \text{(each task assigned to one worker)} \tag{19}$$

$$\sum_{j=1}^{n} x_{ij} \leq k_i, \quad \forall i \quad \text{(worker capacity constraint)} \tag{20}$$

$$x_{ij} \in \{0, 1\}, \quad \forall i, j. \tag{21}$$

3.4 Integer Programming in Python

The following Python code demonstrates how to solve this task assignment problem using the `pulp` library:

```
from pulp import LpMaximize, LpProblem, LpVariable, lpSum

# Define problem parameters (example values)
n = 3  # Number of tasks
m = 2  # Number of workers
c = [[10, 8, 12], [7, 9, 15]]  # Cost matrix (worker i, task j)
k = [2, 2]  # Worker capacities (max tasks per worker)

# Define the problem
problem = LpProblem("Task_Assignment", LpMaximize)

# Decision variables
x = [[LpVariable(f"x_{i}_{j}", cat='Binary') for j in range(n)] for i in
 ↪  range(m)]

# Objective function (maximize total cost)
problem += lpSum(c[i][j] * x[i][j] for i in range(m) for j in range(n)),
 ↪  "Total_Cost"

# Constraints
for j in range(n):
    problem += lpSum(x[i][j] for i in range(m)) == 1, f"Task_{j}_Assigned"  #
     ↪  Each task assigned to one worker

for i in range(m):
    problem += lpSum(x[i][j] for j in range(n)) <= k[i], f"Worker_{i}_Capacity"
     ↪  # Worker capacity constraints

# Solve the problem
```

```
problem.solve()

# Output results
print("Status:", problem.status)  # Print solver status
print("Optimal Value:", problem.objective.value())  # Print optimal objective
 ↪  value

for i in range(m):
    for j in range(n):
        if x[i][j].varValue == 1:
            print(f"Task {j + 1} assigned to Worker {i + 1}")
```

Integer programming is widely applicable in scheduling, resource allocation, and logistics, where decisions must be made in discrete units. Understanding IP serves as a stepping stone to more complex combinatorial optimization problems.

4 Quadratic Programming

Quadratic Programming (QP) extends Linear Programming by incorporating quadratic terms into the objective function, allowing for more sophisticated modeling of real-world problems. Unlike LP, where the objective function is strictly linear, QP problems involve an objective function that includes quadratic terms while maintaining linear constraints. These problems frequently arise in finance, engineering, and machine learning applications such as portfolio optimization and support vector machines.

4.1 Formulation of Quadratic Programming Problems

A standard Quadratic Programming problem is formulated as:

$$\min \quad \frac{1}{2}x^T Q x + c^T x \tag{22}$$

subject to:

$$Ax \leq b, \quad x \geq 0, \tag{23}$$

where:

- $x \in \mathbb{R}^n$ is the vector of decision variables.
- $Q \in \mathbb{R}^{n \times n}$ is a symmetric positive semi-definite matrix defining the quadratic terms.
- $c \in \mathbb{R}^n$ represents the linear coefficients in the objective function.
- $A \in \mathbb{R}^{m \times n}$ is the constraint matrix.
- $b \in \mathbb{R}^m$ is the constraint vector.

When Q is positive semi-definite, the problem remains convex, ensuring that any local minimum is also a global minimum.

4.2 Solution Methods for Quadratic Programming

Several methods exist for solving QP problems efficiently:

- **Interior-Point Methods:** Utilize the structure of the feasible region to traverse the interior and efficiently converge to an optimal solution.
- **Active-Set Methods:** Solve the problem iteratively by focusing on a subset of active constraints at each step, making them suitable for sparse problems.
- **Gradient-Based Methods:** Include techniques such as gradient descent and conjugate gradient methods, particularly useful for large-scale problems.

4.3 Example: Portfolio Optimization

Consider an investor seeking to allocate funds across n assets to minimize risk while achieving a desired return. The risk is measured using a covariance matrix Q, and the returns are represented by a vector c. The mathematical formulation is:

$$\min \quad \frac{1}{2}x^T Q x - c^T x \tag{24}$$

$$\text{subject to:} \quad \sum_{i=1}^{n} x_i = 1, \quad x_i \geq 0, \quad \forall i. \tag{25}$$

4.4 Quadratic Programming in Python

We can solve the portfolio optimization problem using Python's `cvxopt` library:

```
from cvxopt import matrix, solvers

# Define Q and c matrices
Q = matrix([[2.0, -1.0], [-1.0, 4.0]])  # Quadratic coefficients
c = matrix([-3.0, -2.0])  # Linear coefficients

# Inequality constraints Gx <= h
G = matrix([[1.0, 1.0], [-1.0, 0.0], [0.0, -1.0]])
h = matrix([2.0, 0.0, 0.0])

# Solve the QP problem
sol = solvers.qp(Q, c, G, h)

# Output results
print("Optimal Solution:", sol['x'])
print("Optimal Value:", sol['primal objective'])
```

Quadratic programming is a powerful extension of LP, enabling more refined decision-making in optimization problems that involve interactions between variables, such as risk minimization and machine learning model tuning.

5 Introduction to Dynamic Programming

Dynamic programming offers an alternative to solving optimization problems that constructs solutions to problems utilizing a method that is similar to two previous methods discussed in this course:

- **Divide-and-conquer:** Solutions are constructed by solving subproblems and then combining to de- termine the solution to the global problem. This idea is used in dynamic programming in that we require optimal solutions to subproblems when we make choices at the global level. Dynamic programming differs from divide-and-conquer in that dynamic programming algorithms usually do not solve problems recursively because subproblems may be considered multiple times (although, they can be made recursive by using a technique called “memoization").
- **Greedy choice:** Solutions are constructed by making locally optimal choices under the assumption that these choices are independent of previous solutions. Greedy strategies depend upon the principle of optimality in that they depend on previous local choices being optimal. This idea is used in dynamic programming in that we still make a greedy choice among a set of alternatives. Dynamic programming differs from the greedy strategy in that, based on the way the subproblems are created, the decision are no longer local.

Typically, a dynamic programming algorithm is defined by following four steps:

1. Characterize the structure of an optimal solution (applying the principle of optimality).
2. Recursively define the value of an optimal solution (i.e., define the Bellman equation).
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from the computed value function (i.e., Bellman equation).

Given this procedure, we have two key ingredients to dynamic programming problems—optimal substructure and overlapping subproblems. Optimal substructure arises when an optimal solution to a problem contains within it the optimal solutions to its subproblems. The algorithm must then consider a set of possible “splits" of the problem (like divide-and-conquer) and choose the best split based on the values of those subproblems. Since we must select the appropriate split, this suggests there exists overlap among the subproblems such that the split cannot be made without considering those values. As a result, it is preferable that the space of subproblems be kept small (i.e., polynomial in the input size) to manage complexity. Also, because of the overlap of subproblems, a simple recursive approach naturally leads to resolving previously-solved problems unless information obtained from previous solutions is stored in a table for simple lookup. In fact, it is customary to solve dynamic programming in a bottom-up fashion to build this table and avoid the problems arising from recursion.

The main idea that captures both optimal substructure and overlapping subproblems is a mathematical form of the value function for a given subproblem. This mathematical form was first suggested by Richard Bellman in 1957 when Bellman invented the dynamic programming method. As a result, this mathematical form has since become known as the "Bellman form" or the “Bellman equation." A typical form of a Bellman equation is as follows:

$$V_{ij} = \min_{i \leq k < j} \{c_{ijk} + V_{ik} + V_{kj}\}$$

In this form, note that we are minimizing (or maximizing) over a choice of split defined by the value k. The actual value function takes into account the value of optimal subproblem solutions V_{ik} and V_{kj} and adds into those values the current cost of the decision c_{ijk}.

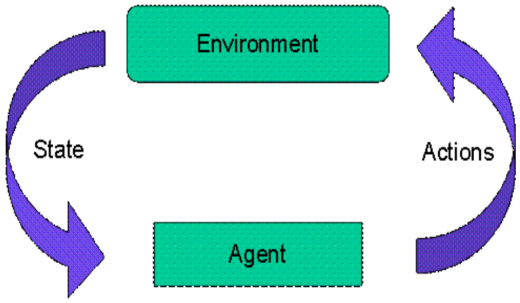
Note that the actual Bellman form can vary substantially from this general form, and the challenge to finding dynamic programming solutions to problems is coming up with this equation. Once the equation is found, the algorithm itself generally follows in a straightforward fashion.

6 Markov Decision Processing

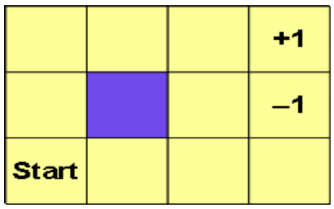
The following is based on material extracted from Barto, Bradtke, and Singh, "Learning to Act Using Real-Time Dynamic Programming," *Artificial Intelligence*, 72 (1): 81–138, 1995. This presentation focuses on the formulation of the original control problem by Richard Bellman and his approaches to solving them. Specifically, the Markov Decision Process was the basis for the development of the dynamic programming method and can be mapped to almost every dynamic programming problem.

6.1 Agent-Based Systems

At the foundation of the Markov Decision Process is the desire to solve control and planning problems for autonomous agents. An agent is defined to be a system (whether or hardware, software, biological, or whatever) that perceives the state of its environment and acts upon those perceptions in an attempt to achieve some goal. This definition is depicted graphically as follows:



For example, suppose we have a robot that want to navigate a simple environment defined by 12 blocks:



In this environment, the starting location for the robot is indicated by "Start", and two ending locations are specified by the values +1 and -1. These values correspond to a "payoff" that the robot will receive when it reaches either location. All other locations have no return, but movement

incurs a fixed cost (say -0.1). The blue square in the center of the environment indicates an obstacle that cannot be occupied by the robot. The robot is able to move up, down, left, or right, and if the robot bumps into a wall or obstacle, it stays in the original location. The object is for the robot to find a path from Start to one of the terminal states and maximize its return (or equivalently minimize its loss).

Two different versions of this problem can be presented. The first is virtually identical to the shortest path problem we saw last week when considering graph algorithms. In this "deterministic" version of the problem, all of the actions yield the expected behavior. In other words, if the robot specifies that it wants to go up, it will go up (as long as it does not hit a wall), and so on for all of the available actions. The second, "nondeterministic" version is the same as the deterministic, except there is noise in the actions. For example, we might find that when a direction is specified, the robot succeeds in moving that direction only 80% of the time. It would move to the left of what was expected 10% of the time and to the right of what was expected 10% of the time. Such nondeterminism makes finding an optimal strategy more difficult than simply applying the greedy approach from Dijkstra's algorithm.

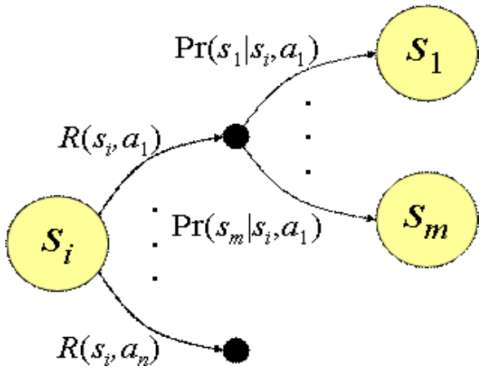
6.2 Definition of an MDP

A Markov Decision Process (MDP) is defined formally as follows:

Definition: M = $\langle$ S, A, T, R, γ $\rangle$ is a Markov Decision Process where

- S is a finite set of states of the environment,
- A is a finite set of actions available to the agent,
- $T : S \times A \rightarrow \pi(S)$ is a state transition function where $T(s,\ a,\ s') = P(s'|s,\ a)$ is the probability of transitioning to state s' given the agent takes action a in state s.
- $R : S \times A \rightarrow \Re$ is a reward function where $R(s,\ a)$ is the expected reward associated with taking action a in state s.
- $\gamma \in [0,\ 1]$ is a discount factor.

This definition can be shown graphically as follows:



It is interesting to note that the assumptions of S and A being finite can be relaxed. In fact, solving MDPs with infinite state and action spaces is an active area of research.

Definition: A policy $\pi : S \rightarrow A$ is a specification of how an agent will act in all states. A policy can be either stationary or nonstationary.

Definition: A stationary policy is a policy that specifies for each step an action to be taken, independent of the time step in which the state is encountered.

Definition: A nonstationary policy is a policy that specifies a sequence of state-action mappings, indexed by time. Specifically, given $\delta = \langle \pi_t, \ldots, \pi_{t+\tau} \rangle$ π_i is used to choose an action in step i as a function of that state.

6.3 Determining the Value of a Policy

Suppose we are presented with a model of an MDP and an associated stationary policy. Then the problem we want to solve is to determine the value of that policy with respect to the MDP. In other words, we want to know what the expected, long run return is if we follow that policy from some state. To find this, let $V^*(s)$ be the expected, discounted, future reward when starting in state s and following stationary policy π. We use the discount factor to ensure this value is well defined; otherwise, an infinite path would yield an infinite cost or reward. Given this, we can find the infinite horizon value of the policy from s as follows:

$$V^*(s) = R(s, \pi(s) + \gamma \sum_{\forall s' \in S} T(s, \pi(s), s') V^*(s').$$

Interpreting, we are saying that the value of state s when following policy can be determined by adding the expected value of the successor state to the reward received when taking action $\pi(s)$ in state s. This is an expected value because the transitions are nondeterministic.

On the surface, this does not appear to help. We have a recurrence that needs to be solved, except we can observe that we can construct a system of |S| linear equations. The only unknowns are the values $V(s)$; therefore, given these equations, we can solve for $V(s)$ using Gaussian elimination or some similar method of solving systems of simultaneous equations.

6.4 Finding an Optimal Policy

Suppose now we are presented with an MDP and know the value of its value function based on some unknown optimal policy. Now the problem becomes extracting the policy from this information. As it turns out, this is a relatively simple thing to do. Specifically, the optimal policy involves applying a greedy choice to the value function, namely

$$\pi(s) = arg \max_{\forall a \in A} [R(s, a) + \gamma \sum_{\forall s' \in S} T(s, a, s') V(s')].$$

6.5 Solving MDPs

Each of the previous two problems had straightforward solutions that did not require dynamic programming. However, suppose we are presented with a situation where we are given the model but do not know either the optimal value function (thus we cannot extract the policy) or the

optimal policy (thus we cannot extract the value function). Is there a way to process this model to find both? The answer is "yes," and that is where dynamic programming comes in. In fact, it can be shown that, for any infinite-horizon, discounted MDP as defined here, there exists a stationary policy that is optimal for every state in the MDP. The value function (i.e., the Bellman equation) for finding that policy is as follows:

$$V(s) = \max_{\forall a \in A} [R(s,\ a) + \gamma \sum_{\forall s' \in S} T(s,\ a,\ s')\, V^*(s')].$$

As we can see, this equation is very similar to the one used to extract the policy, and once we find V, we will be able to use that approach to extract the policy. However, we still need to find V. Unfortunately, we cannot use Gaussian elimination because these equations are no longer linear. To solve, let $V^*(i)$ denote the expected value of the infinite-horizon, discounted reward that will accrue from initial state i given policy π. Let $E_\pi[\cdot]$ denote the expectation over policy π, and let us define $V^*(i)$ as follows:

$$V^*(i) = E_\pi[\textstyle\sum_{t=0}^{\infty} \gamma^t R_t | s_0 = i].$$

Now let's define an intermediate function to accumulate values into a Q table (which will be a cost table for dynamic programming) to approximate V. This can be done as follows:

$$Q^V(i,\ a) = R(i,\ a) + \gamma \sum_{\forall j' \in S} T(i,\ a,\ j)\, Vj).$$

Q represents our approximation of the value of taking action a in state i and then acting optimally thereafter. (As an aside, it is interesting to know that if our MDP is deterministic, the above equation reduces to $Q^V(i,\ a) = R(i,\ a) + \gamma V(j)$.) Since the optimal policy is one that is greedy with respect to the value function, we can rewrite the Bellman equation given above in terms of Q as follows:

$$V(i) = \max_{\forall a \in A} [R(i,\ a) + \gamma \sum_{\forall j \in S} T(i,\ a,\ j)\, V(j)] = \max_{\forall a \in A} Q^V(i,\ a).$$

Given this definition, we are now in a position to define two different dynamic programming algorithms — value iteration and policy iteration.

6.6 Value Iteration

We will compute a sequence V_t using the auxiliary Q function, but we will denote the Q function as $Q_t(s,a)$ to make explicit the relationship between the time step of the algorithm (i.e., the iteration through the loop) and the states and actions in the MDP. Since these Q values change with each step of the algorithm (and so do the corresponding V values), we will be defining a nonstationary policy until convergence. Value iteration will terminate when the maximum difference between two successive iterations drops below a predefined threshold ϵ, known as the Bellman error magnitude. The pseudocode for value iteration is as follows:

The first thing that happens is that all values of V are initialized to zero. Then the main loop is run and continues until the termination criterion described above is met. For this algorithm, every state-action pair is considered and updated using the equation $Q_t(s,\ a) = R(s,\ a) + \gamma(\sum_{\forall s' \in S}) T(s, a, s')V_{t-1}(s')$. After the Q values are updated, the policy is determined by applying the greedy choice $\pi_t(s) = \arg\max_{\forall a \in A} Q_t(s, a)$, and this is used to determine the new value for $V_t(s)$.

Algorithm 1 Value Iteration

for each: $s \in \mathcal{S}$ do
  $V_0 \leftarrow 0$
  $t \leftarrow 0$
  repeat
    $t \leftarrow t + 1$
for each: $s \in \mathcal{S}$ do
for each: $a \in \mathcal{A}$ do
    $Q_t(s, a) \leftarrow R(s, a) + \gamma \sum_{s' \in S} T(s, a, s')V_{t-1}(s')$
    $\pi_t(s) \leftarrow \arg\max_{a \in A} Q_t(s, a)$
    $V_t(s) \leftarrow Q_t(s, \pi_t(s))$
  until $\max_s |V_t(s - V_{t-1}(s)| < \epsilon$
  return π_t

6.7 Policy Iteration

Policy iteration is similar to value iteration in that the intermediate Q function is the basis for the updates; however, this approach uses the two algorithms presented at the beginning of this section (deriving a value function from a policy and deriving a policy given a value function) more directly.

Let π_0 be the greedy policy for the initial value function V_0. Let $V_0 = \mathrm{V}^*$ be the value function for π_0. If we alternate between finding the optimal policy for a particular value function and the corresponding value function for the new policy, we eventually converge on the optimal policy and value function for the MDP. This time, since the value function is derived directly from a policy, the termination criterion for the algorithm is stronger. We terminate when there is no change in the value function because there is no change in the policy. The pseudocode for policy iteration is as follows:

Algorithm 2 Policy Iteration

for each: $s \in \mathcal{S}$ do
  $V_0 \leftarrow 0$
  $t \leftarrow 0$
  repeat
    $t \leftarrow t + 1$
for each: $s \in \mathcal{S}$ do
for each: $a \in \mathcal{A}$ do
    $Q_t(s, a) \leftarrow R(s, a) + \gamma \sum_{s' \in S} T(s, a, s')V_{t-1}(s')$
    $\pi_t(s) \leftarrow \arg\max_{a \in A} Q_t(s, a)$
    $V_t(s) \leftarrow$ (EvalMDP)(π_t, MDP)
  until $V_t(s - V_{t-1}(s)$ for all $s \in S$
  return π_t

The primary difference between value iteration and policy iteration lies in the line $V_t(s) \leftarrow$ EvalMDP(π_t,MDP). This function solves the system of simultaneous equations to determine V directly from the policy.

6.8 Convergence of Synchronous Dynamic Programming

Both value iteration and policy iteration are called "synchronous" dynamic programming because updates at time t are all based on values from time t - 1. Determining the complexity of synchronous dynamic programming (whether policy iteration or value iteration) is difficult for MDPs in that the analysis depends on determining when the process converges. We note that, given n states and m actions, each iteration of the algorithm will require $O(mn^2)$ operations. Policy iteration then has the expense associated with solving the system of simultaneous equations; however, since each $V(s)$ is defined in terms of only one $V(s')$, solving the equations can be done in linear time. In fact, this process is called "backing up" since we effectively back up values through the stages of the MDP, one stage at a time with each iteration.

For convergence, we find that synchronous dynamic programming converges to the optimal V^* in undiscounted stochastic MDPs (i.e., when $\gamma = 1$) under the following conditions:

- The initial cost of every goal state is zero.
- There exists at least one proper policy.
- All policies that are not proper incur infinite cost for at least one state.

Convergence results with similar conditions apply for discounted MDPs as well.

7 Heuristic and Metaheuristic Methods

Optimization problems, especially those involving combinatorial or nonlinear constraints, often become computationally intractable for exact methods. In such cases, heuristic and metaheuristic methods provide efficient approximations to optimal solutions. These methods are inspired by natural and real-world processes, leveraging strategies from biology, physics, and social behavior to navigate complex search spaces.

7.1 Heuristic Methods

Heuristic methods are problem-specific techniques designed to find good, but not necessarily optimal, solutions efficiently. They exploit domain knowledge and predefined rules to guide the search process. While they lack guarantees of optimality, they are widely used in practical applications where speed and feasibility are more critical than absolute precision.

Greedy Algorithm: A simple heuristic that builds a solution incrementally by making locally optimal choices at each step. While efficient, it may not always yield the global optimum. Examples include:

- Shortest path problems using Dijkstra's algorithm.
- Huffman coding for data compression.

Local Search: Iteratively improves a candidate solution by exploring its neighbors. When no further improvement is possible, the algorithm terminates. Variants include:

- Hill climbing: Moves to the best neighbor but can get stuck in local optima.
- Simulated annealing: Introduces random jumps to escape local optima (described below).

7.2 Metaheuristic Methods

Metaheuristics are higher-level strategies that guide heuristic methods in exploring the solution space. They are often inspired by natural processes and provide mechanisms to balance exploration (diversifying search) and exploitation (refining promising solutions).

7.2.1 Simulated Annealing (SA)

Simulated Annealing (SA) is inspired by the annealing process in metallurgy, where a material is heated and then slowly cooled to form a low-energy crystalline structure. In optimization, this process is mimicked by:

- Initializing with a high "temperature" and a random solution.
- Introducing random perturbations to explore the solution space.
- Accepting worse solutions with a probability governed by the Boltzmann distribution:

$$P = e^{-\Delta E/T}, \tag{26}$$

  where ΔE is the change in objective value and T is the current temperature.
- Gradually lowering the temperature to refine the solution and converge to an optimum.

SA is particularly useful in escaping local optima and works well in problems with rugged search landscapes.

7.2.2 Genetic Algorithms (GA)

Genetic Algorithms (GA) draw inspiration from Darwinian evolution, using principles of natural selection to evolve optimal solutions. The process involves:

- **Selection:** Choosing the fittest individuals based on their objective function values to pass on genetic material.
- **Crossover:** Combining two parent solutions to generate new offspring solutions with traits from both parents.
- **Mutation:** Introducing small random modifications to offspring to maintain genetic diversity and prevent premature convergence.

GA is widely used in engineering, scheduling, and machine learning applications such as hyperparameter optimization.

7.2.3 Particle Swarm Optimization (PSO)

Particle Swarm Optimization (PSO) is inspired by the collective intelligence observed in bird flocks and fish schools. It models a population of particles that move through the solution space, each adjusting its position based on:

- Its own best-known position (personal best).
- The best-known position found by any particle (global best).
- An inertia component that helps maintain momentum.

The velocity update rule is given by:

$$v_i^{t+1} = \omega v_i^t + c_1 r_1 (p_i - x_i) + c_2 r_2 (g - x_i), \tag{27}$$

where ω is inertia, c_1, c_2 are acceleration coefficients, r_1, r_2 are random values, and p_i, g represent personal and global best positions, respectively.

PSO is particularly effective in continuous optimization problems such as neural network training and parameter tuning.

7.2.4 Ant Colony Optimization (ACO)

Ant Colony Optimization (ACO) is inspired by the foraging behavior of ants, where ants deposit pheromones to reinforce paths leading to food sources. The algorithm works as follows:

- A population of artificial ants explores possible solutions.
- Pheromone levels on paths are updated to reflect the quality of solutions found.
- Ants probabilistically choose paths based on pheromone intensity and heuristic desirability.
- Over iterations, pheromone accumulation leads to convergence toward optimal solutions.

ACO is particularly effective in graph-based optimization problems such as network routing, vehicle routing, and the traveling salesman problem.

7.3 Comparison and Practical Use Cases

Metaheuristic methods vary in effectiveness based on the problem characteristics:

- **Simulated Annealing:** Suitable for landscapes with many local optima where gradual refinement improves convergence.
- **Genetic Algorithms:** Useful for problems requiring diverse exploration and combination of solutions, such as evolutionary design.
- **Particle Swarm Optimization:** Excels in continuous function optimization, particularly in large-dimensional spaces.
- **Ant Colony Optimization:** Ideal for combinatorial problems like routing and scheduling.

These metaheuristic techniques offer powerful alternatives to exact methods, making them invaluable for large-scale optimization problems where computational efficiency and near-optimal solutions are crucial.

8 Conclusion

Optimization is a cornerstone of decision-making and computational modeling, offering a structured approach to solving complex problems across diverse domains. Throughout these lecture notes, we have explored various optimization techniques, ranging from exact methods such as Linear Programming (LP), Integer Programming (IP), and Quadratic Programming (QP) to approximate metaheuristic methods like Simulated Annealing (SA), Genetic Algorithms (GA), Particle Swarm Optimization (PSO), and Ant Colony Optimization (ACO).

Exact methods provide mathematically rigorous solutions with guarantees of optimality but may become computationally expensive as problem size and complexity increase. In contrast, approximate methods, inspired by natural processes and probabilistic search strategies, offer powerful alternatives that trade off optimality for efficiency and scalability. These heuristic and metaheuristic techniques enable us to tackle large-scale, nonconvex, and combinatorial problems that would otherwise be infeasible with exact solvers.

The relevance of optimization extends beyond theoretical constructs, playing a crucial role in real-world applications such as logistics, finance, engineering design, artificial intelligence, and machine learning. As data-driven decision-making continues to evolve, optimization remains a fundamental tool for refining models, improving efficiency, and driving innovation. Mastering these techniques empowers practitioners to formulate, analyze, and solve optimization problems effectively, making it an indispensable skill in computational sciences.

References

[1] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronal L., and Stein, Clifford, *Introduction to Algorithms*, 3rd Edition, MIT Press, 2009

[2] Honner, Patric (Contributing Columnist), textitWhy Winning in Rock-Paper-Scissors (and in Life) Isn't Everything, What does John Nash's game theory equilibrium concept look like in Rock-Paper-Scissors?, an article in the online Quanta Magazine, April 2, 2018, https://www.quantamagazine.org/the-game-theory-math-behind-rock-paper-scissors-20180402/

[3] David A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", Proceedings of the I.R.E., September 1952, pp 1098-1102.

[4] Background story: Profile: David A. Huffman, Scientific American, September 1991, pp. 54-58.

[5] Wikipedia, "Huffman Coding", Retrived Sept 2009, http://en.wikipedia.org/wiki/Huffman_coding.