AI505

Optimization

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

List of Lectures

1. Introduction
2. Derivatives and Gradients
3. Bracketing
4. Local Descent
5. First-Order Methods
6. Second-Order Methods
7. Direct Methods
8. Stochastic Methods
9. Population Methods
10. Machine Learning Applications
11. Convergence Analysis of SG
12. Beyond Stochastic Gradient
13. Constrained Optimization
14. Linear Programming
15. Sampling Plans
16. Integer Linear Programming
17. Constraint Programming and Heuristics
18. Metaheuristics

1. Introduction

Who is here?

42 in total registered in ItsLearning

- **AI505 (7.5 ECTS)**
  34 from Bachelor in AI

- **AI505 (7.5 ECTS) + IAAI501 (2.5 ECTS)**
  8 from Master in Mathematics and Economics

Prerequisites

- Calculus
- Linear Algebra
- Programming

Outline

1. Course Organization

2. Introduction

Aims of the course

Learn about **optimization**:

- continuous multivariate optimization
- discrete optimization

Optimization is an important tool in **machine learning**, **decision making** and in analysing physical systems.
In mathematical terms, an **optimization problem** is the problem of finding the **best solution** from the set of all **feasible solutions**.

The first step in the optimization process is constructing an appropriate **mathematical formulation**. The second is devising an **algorithm** for solving the mathematical formulation.

Contents of the Course (Pensum)

Unit	Main topic
1	Introduction, Univariate Problems
2	Multivariate Problems, Gradient-Based Methods
3	Derivative-Free Methods
4	Optimization for Machine Learning
5	Constrained Optimization, Linear Programming
6	Sampling Methods
7	Discrete Optimization and Heuristics

Practical Information

Teacher: Marco Chiarandini (imada.sdu.dk/u/marco/)
Instructor: Bonnie Liefting (H21)

Schedule, alternative views:

- mitsdu.sdu.dk, SDU Mobile
- Official course description (læserplanen)
- ItsLearning
- https://ai-505.github.io

Schedule (16 weeks):

- Introductory classes: 40 hours (20 classes)
- Training classes: 30 hours (15 classes)
- Scheduled: $16 \times 4 = 64$ hours
- No classes in week 8

Communication Means

- ItsLearning $\Leftrightarrow$ External Web Page
  (link `https://ai-505.github.io`)
- Announcements in ItsLearning
- Write to Marco (`marco@imada.sdu.dk`) or to instructor
- Collaborate with peers

$\rightsquigarrow$ **It is good to ask questions!!**

$\rightsquigarrow$ **Let me know if you think we should do things differently!**

Sources — Reading Material

Main reference:

[KW] Mykel J. Kochenderfer, Tim A. Wheeler. Algorithms for Optimization 2019. The MIT Press.

Others

[NW] J. Nocedal and S. J. Wright, Numerical Optimization, Second Edition. Springer Series in Operations Research, 2006

Course Material

External Web Page is the main reference for list of contents (ie[1], syllabus, pensum).

It will collect:

- slides
- list of topics and references
- exercises
- links
- tutorials for programming tasks

[1] ie = id est = that is, eg = exempli gratia = for example, wrt = with respect to, et al. = et alii = and others

Assessment

Portfolio consisting of:

- mandatory assignments in groups of 2:
  1. assignment
  2. assignment
  3. assignment
  4. assignment

  Time line to be announced.

- oral exam on June 11-13, 2025

The oral examination consists of questions on the basis of the handed in assignments and can be extended to other parts of the course curriculum.

Final grade: starting from the grade of the assignments overall evaluation can move up or down by at most two levels.

(language: Danish and/or English)

Exercise Sessions (1/2)

For exercises in general:

- Both theoretical, modelling and programming tasks
- No mandatory hand-ins, voluntary participation, but all lectures and exercises are relevant for to exam
- Help each other! Teaching others is the best form of learning
- Bonnie will be there to help
- Exercises should help you to learn - don't hesitate to find a work setting or extend with other material that works better for you

Exercises are group work

- Exercises best done in pairs of 2-3 people
- Try to gather in different groups every now and then

Exercise Sessions (1/2)

Notation: $^{+}$, *, ' '

- plus exercises are to be done before the class
- starred exercises are done in class
- unmarked exercises are for self study
- starred exercises are examples of assignment questions

Coding

Set up your local Python programming environment and use the available tutorial to:

- Brush up some knowledge on Python-IDEs and Git
- Grasp some basics on jupyter notebooks
- Recap basics of data processing and visualization

We will span several programming modes, functional, imperative, object oriented, small scripts large code bases, use pf modules and packages. We will use git.

Outline

1. Course Organization

2. Introduction

Introduction

- Applications of Optimization
  - Physics
  - Business
  - Biology
  - Engineering
  - Machine Learning
  - Logistics and Scheduling
- Objectives to Optimize
  - Efficiency
  - Safety
  - Accuracy
- Constraints
  - Cost
  - Weight
  - Structural Integrity
- Challenges
  - High-Dimensional Search Spaces
  - Multiple Competing Objectives
  - Model Uncertainty

A Brief History

- Algebra, study of the rules for manipulating mathematical symbols
- Calculus, study of continuous change It stems from the developments of Leibniz (1646–1716) and Newton (1642–1727). Both differential and integral calculus make use of the notion of convergence of infinite series to a well-defined limit.
- Computers mid-twentieth and numerical algorithms for optimization
- Linear programming, which is an optimization problem with a linear objective function and linear constraints. Leonid Kantorovich (1912–1986) presented a formulation for linear programming and an algorithm to solve it.
- It was applied to optimal resource allocation problems during World War II. George Dantzig (1914–2005) developed the simplex algorithm, which represented a significant advance in solving linear
- Richard Bellman (1920–1984) developed the notion of dynamic programming, which is a commonly used method for optimally solving complex problems by breaking them down into simpler problems
- Artificial Intelligence ($\Longleftrightarrow$ Optimization)

Real World vs Model vs Representation vs Implementation

The Real World: That messy thing we are trying to study (with computers).

Model: Mathematical object in some class M.

Representation: An object of an abstract data type R used to store the model.

Implementation: An object of a concrete type used to store the model.

Any object from the real world might have different models.
Any model might have several representations (exact).
And representation might have different implementations (exact).

We will focus on the algorithmic aspects of optimization that arise after the problem has been properly formulated

Optimization Process

An optimization algorithm is used to incrementally improve the design until it can no longer be improved or until the budgeted time or cost has been reached.

Optimization Process

Search the space of possible designs with the aim of finding the best one.

Depending on the application, this search may involve:

- evaluating an analytical expression (white or glass box)
- running physical experiments, such as wind tunnel tests (black box)
- running computer simulations

Modern optimization techniques can be applied to problems with millions of variables and constraints.

Basic Optimization Problem

$$\underset{\boldsymbol{x}}{\text{minimize}}\ f(\boldsymbol{x})$$
$$\text{subject to } \boldsymbol{x} \in \mathcal{X}$$

- Design Point
- Design Variables
- Objective Function
- Feasible Set
- Minimizer

Any value of $\boldsymbol{x}$ from among all points in the feasible set $\mathcal{X}$ that minimizes the objective function is called a **solution** or **minimizer**. A particular solution is written $\boldsymbol{x}^*$.

$$\boldsymbol{x}^* = \text{argmin}\, f(\boldsymbol{x}) \text{ subject to } \boldsymbol{x} \in \mathcal{X}$$

There is only one minimum but there can be many minimizers

maximize $f(\boldsymbol{x})$ subject to $\boldsymbol{x} \in \mathcal{X} \equiv$ minimize $-f(\boldsymbol{x})$ subject to $\boldsymbol{x} \in \mathcal{X}$

Basic Optimization Problem

$$\begin{aligned} \underset{\boldsymbol{x}}{\text{minimize}} \quad & f(\boldsymbol{x}) \\ \text{subject to} \quad & \boldsymbol{x} \in \mathcal{X} \end{aligned}$$

Constraints

$$\begin{aligned} \underset{x_1, x_2}{\text{minimize}} \quad & f(x_1, x_2) \\ \text{subject to} \quad & x_1 \geq 0 \\ & x_2 \geq 0 \\ & x_1 + x_2 \leq 1 \end{aligned}$$

Constraints

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & x > 1 \end{aligned}$$

The problem has no solution.
$x = 1$ would not be feasible.

$x = 1$ would be the solutions to

$$\underset{x}{\text{infimum}}\ f(x) \text{ subject to } x > 1$$

infimum of a subset $\mathcal{X}$ of a partially ordered set $\mathcal{P}$ is the greatest element in P that is less than or equal to each element of $\mathcal{X}$, if such an element exists. Aka, **greatest lower bound**.

Critical Points

Univariate function

global vs local minimum

<u>Def.</u> A point $\boldsymbol{x}^*$ is at a **local minimum** (or is a local minimizer) if there exists a $\delta > 0$ such that $f(\boldsymbol{x}^*) \leq f(\boldsymbol{x})$ for all $\boldsymbol{x}$ with $\|\boldsymbol{x} - \boldsymbol{x}^*\| < \delta$.

Conditions for Local Minima

Univariate objective functions, assuming they are differentiable (Derivatives exist), without constraints

Local minimum: Necessary condition but not sufficient condition:

1. $f'(x^*) = 0$, the first-order necessary condition (FONC)
2. $f''(x^*) \geq 0$, the second-order necessary condition (SONC)

SONC but not FONC

FONC and SONC

FONC and SONC

Conditions for Local Minima

Multivariate objective functions, assuming they are differentiable (gradients and Hessians exist), without constraints

Local minimum: Necessary condition but not sufficient condition:

1. $\nabla f(\boldsymbol{x}^*) = 0$, the first-order necessary condition (FONC)
2. $\nabla^2 f(\boldsymbol{x}^*) \geq 0$, the second-order necessary condition (SONC)

Contour Plots

$$f(x_1, x_2) = x_1^2 - x_2^2$$

can be rendered in a 3D space but convenient to represent it also in 2D showing the lines of constant output value

Taylor Expansion

From the *first fundamental theorem of calculus*,[2] we know that

$$f(x+h) = f(x) + \int_0^h f'(x+a)\,da$$

Nesting this definition produces the Taylor expansion of f about x:

$$\begin{aligned}
f(x+h) &= f(x) + \int_0^h \left(f'(x) + \int_0^a f''(x+b)\,db \right) da \\
&= f(x) + f'(x)h + \int_0^h \int_0^a f''(x+b)\,db\,da \\
&= f(x) + f'(x)h + \int_0^h \int_0^a \left(f''(x) + \int_0^b f'''(x+c)\,dc \right) db\,da \\
&= f(x) + f'(x)h + \frac{f''(x)}{2!}h^2 + \int_0^h \int_0^a \int_0^b f'''(x+c)\,dc\,db\,da \\
&\vdots \\
&= f(x) + \frac{f'(x)}{1!}h + \frac{f''(x)}{2!}h^2 + \frac{f'''(x)}{3!}h^3 + \dots \\
&= \sum_{n=0}^{\infty} \frac{f^{(n)}(x)}{n!}h^n
\end{aligned}$$

Taylor Expansion

Taylor Expansion Multidim

In multiple dimensions, the Taylor expansion about $\mathbf{a}$ generalizes to

$$f(\mathbf{x}) = f(\mathbf{a}) + \nabla f(\mathbf{a})^\top (\mathbf{x} - \mathbf{a}) + \frac{1}{2}(\mathbf{x} - \mathbf{a})^\top \nabla^2 f(\mathbf{a})(\mathbf{x} - \mathbf{a}) + \dots$$

Example: Rosenbrock function

$$f(x, y) = (a - x)^2 + b(y - x^2)^2$$

It has a global minimum at $(x, y) = (a, a^2)$, where $f(x, y) = 0$. Usually, these parameters are set such that $a = 1$ and $b = 100$. Only in the trivial case where $a = 0$ the function is symmetric and the minimum is at the origin.

Multivariate generalization
sum of $N/2$ uncoupled 2D Rosenbrock problems, and defined only for even N:

$$f(\mathrm{x}) = f(x_1, x_2, \ldots, x_N) = \sum_{i=1}^{N/2} \left[100(x_{2i-1}^2 - x_{2i})^2 + (x_{2i-1} - 1)^2\right].$$

This variant has predictably simple solutions.

Example: Rosenbrock function

Consider the Rosenbrock banana function,

$$f(\mathbf{x}) = (1 - x_1)^2 + 5(x_2 - x_1^2)^2$$

Does the point $(1, 1)$ satisfy the FONC and SONC?

The gradient is:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2\left(10x_1^3 - 10x_1x_2 + x_1 - 1\right) \\ 10(x_2 - x_1^2) \end{bmatrix}$$

and the Hessian is:

$$\nabla^2 f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2 \partial x_2} \end{bmatrix} = \begin{bmatrix} -20(x_2 - x_1^2) + 40x_1^2 + 2 & -20x_1 \\ -20x_1 & 10 \end{bmatrix}$$

We compute $\nabla(f)([1, 1]) = 0$, so the FONC is satisfied. The Hessian at $[1, 1]$ is:

$$\begin{bmatrix} 42 & -20 \\ -20 & 10 \end{bmatrix}$$

which is positive definite, so the SONC is satisfied.

Example: Rosenbrock function

Overview

2. Derivatives → 4. Descent → 5. First-Order
3. Bracketing → 6. Second-Order

7. Direct

8. Stochastic → 9. Population → 20. Expressions
10. Constraints → 11. Linear → 19. Discrete
→ 12. Multiobjective

13. Sampling Plans → 14. Surrogate Models → 15. Probabilistic Surrogate Models → 16. Surrogate Optimization

17. Optimization under Uncertainty → 18. Uncertainty Propagation

21. Multidisciplinary Design Optimization

Optimization Problem Types

(NEOS Server, University of Wisconsin)

https://neos-guide.org/guide/types/

Problem classification

Different classification parameters:

- Univariate $f : \mathbb{R} \to \mathbb{R}$ vs Multivariate $f : \mathbb{R}^n \to \mathbb{R}$
- Real-valued $f : \mathbb{R}^n \to \mathbb{R}$ vs vector functions $f : \mathbb{R}^n \to \mathbb{R}^m$
- Linear vs Nonlinear
- Nonlinear: Convex vs Nonconvex, unimodal vs multimodal
- Constrained vs unconstrained
- Smooth (differentiable) vs non smooth (non differentiable)
- Deterministic vs Uncertain
- Continuous vs Discrete

Application Example

In robotics, an autonomous agent (e.g., a drone, robotic arm, or self-driving car) needs to determine an **optimal path** from a starting position to a target while satisfying constraints such as avoiding obstacles, minimizing energy consumption, or optimizing smoothness.
Formulating the Optimization Problem
The goal is to find a smooth trajectory $x(t)$ that minimizes a cost function:

$$J = \int_{t_0}^{t^f} C(\boldsymbol{x}(t), \boldsymbol{u}(t))dt$$

$\boldsymbol{x}(t)$ is the state (e.g., position, velocity), $\boldsymbol{u}(t)$ is the control input (e.g., force, acceleration), $C(\boldsymbol{x}, \boldsymbol{u})$ is the cost function, which could represent energy usage, time, or distance.
Constraints:

- Dynamic Constraints: Governed by the system's physics (e.g., Newton's laws for a robot arm or quadcopter): $\boldsymbol{x}' = f(\boldsymbol{x}, \boldsymbol{u})$
- Obstacle Avoidance: Ensures that the trajectory does not collide with obstacles: $g(\boldsymbol{x}(t)) \geq 0, \forall t$
- Boundary Conditions: The system must start and end at given positions with certain velocities.

Summary

- Optimization is the process of finding the best system design subject to a set of constraints
- Optimization is concerned with finding global minima of a function
- Minima can occur where the gradient is zero, but zero-gradient does not imply optimality

2. Derivatives and Gradients

Definitions

- $[a, b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}$ closed interval
  $(a, b) = \{x \in \mathbb{R} \mid a < x < b\}$ open interval
- column vectors and matrices
  scalar product: $\boldsymbol{y}^T \boldsymbol{x} = \sum_{i=1}^{n} y_i x_i$
- $A\boldsymbol{x}$ column vector combination of the columns of A;
  $\boldsymbol{u}^T A$ row vector combination of the rows of A

Definitions

- **linear combination**

$$\begin{aligned} \boldsymbol{v}_1, \boldsymbol{v}_2 \ldots, \boldsymbol{v}_k &\in \mathbb{R}^n \\ \boldsymbol{\lambda} = [\lambda_1, \ldots, \lambda_k]^T &\in \mathbb{R}^k \end{aligned} \qquad \boldsymbol{x} = \lambda_1 \boldsymbol{v}_1 + \cdots + \lambda_k \boldsymbol{v}_k = \sum_{i=1}^{k} \lambda_i \boldsymbol{v}_i$$

moreover:

$\boldsymbol{\lambda} \geq 0$	**conic combination**	
$\boldsymbol{\lambda}^T 1 = 1$	**affine combination**	$\left(\sum_{i=1}^{k} \lambda_i = 1 \right)$
$\boldsymbol{\lambda} \geq 0$ and $\boldsymbol{\lambda}^T 1 = 1$	**convex combination**	

Definitions

- **convex set**: if $\boldsymbol{x}, \boldsymbol{y} \in S$ and $0 \leq \lambda \leq 1$ then $\lambda \boldsymbol{x} + (1-\lambda)\boldsymbol{y} \in S$

nonconvex

convex

- **convex function**
  - if its epigraph $\{(x, y) \in \mathbb{R}^2 : y \geq f(x)\}$ is a convex set or
  - if $f : \mathbb{R}^n \to \mathbb{R}$ and if $\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n, \alpha \in [0, 1]$ it holds that $f(\alpha \boldsymbol{x} + (1-\alpha)\boldsymbol{y}) \leq \alpha f(\boldsymbol{x}) + (1-\alpha) f(\boldsymbol{y})$

Definitions

- For a set of points $S \subseteq \mathbb{R}^n$

  $\mathrm{lin}(S)$ linear hull (span)
  $\mathrm{cone}(S)$ conic hull
  $\mathrm{aff}(S)$ affine hull
  $\mathrm{conv}(S)$ convex hull

the convex hull of X

$$\mathrm{conv}(X) = \left\{ \lambda_1 x_1 + \lambda_2 x_2 + \ldots + \lambda_n x_n \;\middle|\; x_i \in X,\ \lambda_1, \ldots, \lambda_n \geq 0 \text{ and } \textstyle\sum_i \lambda_i = 1 \right\}$$

Norms

<u>Def.</u> A **norm** is a function that assigns a length to a vector.

A function f is a norm if:

1. $f(\boldsymbol{x}) = 0$ if and only if $\boldsymbol{x}$ is the zero vector
2. $f(a\boldsymbol{x}) = |a| f(\boldsymbol{x})$, such that lengths scale
3. $f(\boldsymbol{x} + \boldsymbol{y}) \leq f(\boldsymbol{x}) + f(\boldsymbol{y})$, also known as triangle inequality

L_p norms are commonly used set of norms parameterized by a scalar $p \geq 1$:

$$\|\boldsymbol{x}\|_p = \lim_{\rho \to p} \left(|x_1|^\rho + |x_2|^\rho + \ldots + |x_n|^\rho\right)^{\frac{1}{\rho}}$$

L_∞ is also called the max norm, Chebyshev distance or chessboard distance.

L_1: $\|\mathbf{x}\|_1 = |x_1| + |x_2| + \cdots + |x_n|$

L_2: $\|\mathbf{x}\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$

L_∞: $\|\mathbf{x}\|_\infty = \max(|x_1|, |x_2|, \cdots, |x_n|)$

Outline

3. Derivatives

4. Symbolic Differentiation

5. Numerical Differentiation

6. Automatic Differentiation

Derivatives

- Derivatives tell us which direction to search for a solution

- Slope of Tangent Line

$$f'(x) := \frac{\mathrm{d}f(x)}{\mathrm{d}x}$$

(Leibniz notation)

Derivatives

$$f(x + \Delta x) \approx f(x) + f'(x)\Delta x$$

$$f'(x) = \frac{\Delta x}{\Delta x}$$

Symbolic Differentiation

$$f'(x) \equiv \underbrace{\lim_{h\to 0} \frac{f(x+h)-f(x)}{h}}_{\text{forward difference}} = \underbrace{\lim_{h\to 0} \frac{f(x+h/2)-f(x-h/2)}{h}}_{\text{central difference}} = \underbrace{\lim_{h\to 0} \frac{f(x)-f(x-h)}{h}}_{\text{backward difference}}$$

Symbolic Differentiation

```python
import sympy as sp

# Define the variable
x = sp.symbols('x')

# Define the function
f = x**2 + x/2 - sp.sin(x)/x

# Compute the derivative
df_dx = sp.diff(f, x)

# Display the result
print("The symbolic derivative of f is:")
print(df_dx)
```

derivative.py

Derivatives in Multiple Dimensions

- Gradient Vector

$$\nabla f(\boldsymbol{x}) = \left[\frac{\partial f(\boldsymbol{x})}{\partial x_1}, \frac{\partial f(\boldsymbol{x})}{\partial x_2}, \ldots, \frac{\partial f(\boldsymbol{x})}{\partial x_n}\right]$$

- Hessian Matrix

$$\nabla^2 f(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial^2 f(\boldsymbol{x})}{\partial x_1\,\partial x_1} & \frac{\partial^2 f(\boldsymbol{x})}{\partial x_1\,\partial x_2} & \cdots & \frac{\partial^2 f(\boldsymbol{x})}{\partial x_1\,\partial x_n} \\ \frac{\partial^2 f(\boldsymbol{x})}{\partial x_1\,\partial x_2} & \frac{\partial^2 f(\boldsymbol{x})}{\partial x_2\,\partial x_2} & \cdots & \frac{\partial^2 f(\boldsymbol{x})}{\partial x_2\,\partial x_n} \\ \vdots & \ddots & & \vdots \\ \frac{\partial^2 f(\boldsymbol{x})}{\partial x_1\,\partial x_n} & \frac{\partial^2 f(\boldsymbol{x})}{\partial x_2\,\partial x_n} & \cdots & \frac{\partial^2 f(\boldsymbol{x})}{\partial x_n\,\partial x_n} \end{bmatrix}$$

Directional derivative

The **directional derivative** $\nabla_{\boldsymbol{s}} f(\boldsymbol{x})$ of a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$ is the instantaneous rate of change of $f(\boldsymbol{x})$ as $\boldsymbol{x} = [x_1, x_2, \ldots, x_n]$ is moved with velocity $\boldsymbol{s} = [s_1, s_2, \ldots, s_n]$.

$$\nabla_{\mathbf{s}} f(\mathbf{x}) \equiv \underbrace{\lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{s}) - f(\mathbf{x})}{h}}_{\text{forward difference}} = \underbrace{\lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{s}/2) - f(\mathbf{x} - h\mathbf{s}/2)}{h}}_{\text{central difference}} = \underbrace{\lim_{h \to 0} \frac{f(\mathbf{x}) - f(\mathbf{x} - h\mathbf{s})}{h}}_{\text{backward difference}}$$

To compute $\nabla_{\boldsymbol{s}} f(\boldsymbol{x})$:

- compute $\boxed{\nabla_{\boldsymbol{s}} f(\boldsymbol{x}) = \frac{\partial f(\boldsymbol{x})}{\partial x_1} s_1 + \frac{\partial f(\boldsymbol{x})}{\partial x_2} s_2 + \ldots + \frac{\partial f(\boldsymbol{x})}{\partial x_n} s_n = \nabla f(\boldsymbol{x})^T \boldsymbol{s} = \nabla f(\boldsymbol{x}) \cdot \boldsymbol{s}}$
- $g(\alpha) := f(\boldsymbol{x} + \alpha \boldsymbol{s})$ and then compute $g'(0)$

We wish to compute the directional derivative of $f(\mathbf{x}) = x_1 x_2$ at $\mathbf{x} = [1, 0]$ in the direction $\mathbf{s} = [-1, -1]$:

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f}{\partial x_1}, \quad \frac{\partial f}{\partial x_2}\right] = [x_2, x_1]$$

$$\nabla_{\mathbf{s}} f(\mathbf{x}) = \nabla f(\mathbf{x})^\top \mathbf{s} = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \end{bmatrix} = -1$$

We can also compute the directional derivative as follows:

$$g(\alpha) = f(\mathbf{x} + \alpha \mathbf{s}) = (1 - \alpha)(-\alpha) = \alpha^2 - \alpha$$

$$g'(\alpha) = 2\alpha - 1$$

$$g'(0) = -1$$

Matrix Calculus

Common gradient:

$$\nabla_{\boldsymbol{x}} \boldsymbol{b}^T \boldsymbol{x} = ?$$

$$\boldsymbol{b}^T \boldsymbol{x} = [b_1 x_1 + b_2 x_2 + \ldots + b_n x_n]$$

$$\frac{\partial \boldsymbol{b}^T \boldsymbol{x}}{\partial x_i} = b_i$$

$$\nabla_{\boldsymbol{x}} \boldsymbol{b}^T \boldsymbol{x} = \nabla_{\boldsymbol{x}} \boldsymbol{x}^T \boldsymbol{b} = \boldsymbol{b}$$

Matrix Calculus

Common gradient:

$$\nabla_{\boldsymbol{x}} \boldsymbol{x}^T A \boldsymbol{x} = ?$$

$$\boldsymbol{x}^T A \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}^T \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}^T \begin{bmatrix} x_1 a_{11} + x_2 a_{12} + \dots + x_n a_{1n} \\ x_1 a_{21} + x_2 a_{22} + \dots + x_n a_{2n} \\ \vdots \\ x_1 a_{n1} + x_2 a_{n2} + \dots + x_n a_{nn} \end{bmatrix}$$

$$= \begin{matrix} x_1^2 a_{11} + x_1 x_2 a_{12} + \dots + x_1 x_n a_{1n} + \\ x_1 x_2 a_{21} + x_2^2 a_{22} + \dots + x_2 x_n a_{2n} + \\ \vdots \\ x_1 x_n a_{n1} + x_2 x_n a_{n2} + \dots + x_n^2 a_{nn} \end{matrix}$$

$$\frac{\partial}{\partial x_i} \boldsymbol{x}^T A \boldsymbol{x} = \sum_{j=1}^{n} x_j \left(a_{ij} + a_{ji}\right)$$

$$\nabla_x \boldsymbol{x}^T A \boldsymbol{x} = \begin{bmatrix} \sum_{j=1}^{n} x_j \left(a_{1j} + a_{j1}\right) \\ \sum_{j=1}^{n} x_j \left(a_{2j} + a_{j2}\right) \\ \vdots \\ \sum_{j=1}^{n} x_j \left(a_{nj} + a_{jn}\right) \end{bmatrix} = \begin{bmatrix} a_{11} + a_{11} & a_{12} + a_{21} & \dots & a_{1n} + a_{n1} \\ a_{21} + a_{12} & a_{22} + a_{22} & \dots & a_{2n} + a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} + a_{1n} & a_{n2} + a_{2n} & \dots & a_{nn} + a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \left(A + A^T\right) \boldsymbol{x}$$

Smoothness

Def. The **smoothness** of a function is a property measured by the number of continuous derivatives (differentiability class) it has over its domain.

A function of **class** C^k is a function of smoothness at least k; that is, a function of class C^k is a function that has a kth derivative that is continuous in its domain.

The term **smooth function** refers to a C^∞-function. However, it may also mean "sufficiently differentiable" for the problem under consideration.

Smoothness

- Let U be an open set on the real line and a function f defined on U with real values. Let k be a non-negative integer.
- The function f is said to be of **differentiability class** C^k if the derivatives $f', f'', \ldots, f^{(k)}$ exist and are continuous on U.
- If f is k-differentiable on U, then it is at least in the class C^{k-1} since $f', f'', \ldots, f^{(k-1)}$ are continuous on U.
- The function f is said to be **infinitely differentiable**, **smooth**, or of **class** C^∞, if it has derivatives of all orders (continuous) on U.
- The function f is said to be of **class** C^ω, or **analytic**, if f is smooth and its Taylor series expansion around any point in its domain converges to the function in some neighborhood of the point.
- There exist functions that are smooth but not analytic; C^ω is thus strictly contained in C^∞. Bump functions are examples of functions with this property.

Example: continuous (C^0) but not differentiable [edit]

The function

$$f(x) = \begin{cases} x & \text{if } x \geq 0, \\ 0 & \text{if } x < 0 \end{cases}$$

is continuous, but not differentiable at $x = 0$, so it is of class C^0, but not of class C^1.

The C^0 function $f(x) = x$ for $x \geq 0$ and 0 otherwise.

Example: finitely-times differentiable (C^k) [edit]

For each even integer k, the function

$$f(x) = |x|^{k+1}$$

is continuous and k times differentiable at all x. At $x = 0$, however, f is not $(k + 1)$ times differentiable, so f is of class C^k, but not of class C^j where $j > k$.

The function $g(x) = x^2 \sin(1/x)$ for $x > 0$.

Example: differentiable but not continuously differentiable (not C^1) [edit]

The function

$$g(x) = \begin{cases} x^2 \sin\left(\frac{1}{x}\right) & \text{if } x \neq 0, \\ 0 & \text{if } x = 0 \end{cases}$$

is differentiable, with derivative

$$g'(x) = \begin{cases} -\cos\left(\frac{1}{x}\right) + 2x \sin\left(\frac{1}{x}\right) & \text{if } x \neq 0, \\ 0 & \text{if } x = 0. \end{cases}$$

Because $\cos(1/x)$ oscillates as $x \to 0$, $g'(x)$ is not continuous at zero. Therefore, $g(x)$ is differentiable but not of class C^1.

The function $f : \mathbb{R} \to \mathbb{R}$ with $f(x) = x^2 \sin\left(\frac{1}{x}\right)$ for $x \neq 0$ and $f(0) = 0$ is differentiable. However, this function is not continuously differentiable.

Example: differentiable but not Lipschitz continuous [edit]

The function

$$h(x) = \begin{cases} x^{4/3} \sin\left(\frac{1}{x}\right) & \text{if } x \neq 0, \\ 0 & \text{if } x = 0 \end{cases}$$

is differentiable but its derivative is unbounded on a compact set. Therefore, h is an example of a function that is differentiable but not locally Lipschitz continuous.

Example: analytic (C^ω) [edit]

The exponential function e^x is analytic, and hence falls into the class C^ω (where ω is the smallest transfinite ordinal). The trigonometric functions are also analytic wherever they are defined, because they are linear combinations of complex exponential functions e^{ix} and e^{-ix}.

A smooth function that is not analytic.

Example: smooth (C^∞) but not analytic (C^ω) [edit]

The bump function

$$f(x) = \begin{cases} e^{-\frac{1}{1-x^2}} & \text{if } |x| < 1, \\ 0 & \text{otherwise} \end{cases}$$

is smooth, so of class C^∞, but it is not analytic at $x = \pm 1$, and hence is not of class C^ω. The function f is an example of a smooth function with compact support.

Positive Definteness

<u>Def.</u> A symmetric matrix A is **positive definite** if $\boldsymbol{x}^T A\boldsymbol{x}$ is positive for all points other than the origin: $x^T A\boldsymbol{x} > 0$ for all $x \neq 0$.
<u>Def.</u> A symmetric matrix A is **positive semidefinite** if $\boldsymbol{x}^T A\boldsymbol{x}$ is always non-negative: $\boldsymbol{x}^T A\boldsymbol{x} \geq 0$ for all $\boldsymbol{x}$.

A matrix A is positive definite if and only all its eigenvalues are positive.

If the matrix A is positive definite in the function $f(\boldsymbol{x}) = x^T A\boldsymbol{x}$, then f has a unique global minimum.

Recall that the second order Taylor approximation of a twice-differentiable function f at x_0 is

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}_0) + \nabla f(\boldsymbol{x}_0)^T(\boldsymbol{x} - \boldsymbol{x}_0) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_0)^T H_0(\boldsymbol{x} - \boldsymbol{x}_0)$$

where H_0 is the Hessian evaluated at $\boldsymbol{x}_0$. If $(\boldsymbol{x} - \boldsymbol{x}_0)^T H_0(\boldsymbol{x} - \boldsymbol{x}_0)$ has a unique global minimum, then the overall approximation has a unique global minimum.

Outline

3. Derivatives

4. Symbolic Differentiation

5. Numerical Differentiation

6. Automatic Differentiation

Symbolic Derivatives

- Symbolic derivatives can give valuable insight into the structure of the problem domain and, in some cases, produce analytical solutions of extrema (e.g., solving for $\frac{\mathrm{d}}{\mathrm{d}x} f(x) = 0$) that can eliminate the need for derivative calculation altogether.

- But they do not lend themselves to efficient runtime calculation of derivative values, as they can get exponentially larger than the expression whose derivative they represent

Outline

3. Derivatives

4. Symbolic Differentiation

5. Numerical Differentiation

6. Automatic Differentiation

Numerical Differentiation

Finite Difference Method

- Neighboring points are used to approximate the derivative

$$f'(x) \approx \underbrace{\frac{f(x+h)-f(x)}{h}}_{\text{forward difference}} \approx \underbrace{\frac{f(x+h/2)-f(x-h/2)}{h}}_{\text{central difference}} \approx \underbrace{\frac{f(x)-f(x-h)}{h}}_{\text{backward difference}}$$

- h too small causes numerical cancellation errors (square root or cube root of the machine precision for floating point values: `sys.float_info.epsilon` difference between 1 and closest representable number)

Derivation

from Taylor series expansion:

$$f(x+h) = f(x) + \frac{f'(x)}{1!}h + \frac{f''(x)}{2!}h^2 + \frac{f'''(x)}{3!}h^3 + \cdots$$

We can rearrange and solve for the first derivative:

$$f'(x)h = f(x+h) - f(x) - \frac{f''(x)}{2!}h^2 - \frac{f'''(x)}{3!}h^3 - \cdots$$

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{f''(x)}{2!}h - \frac{f'''(x)}{3!}h^2 - \cdots$$

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

- forward difference has error term $O(h)$, linear error as h approaches zero
- central difference has error term is $O(h^2)$

```python
import sys
import numpy as np

def diff_forward(f, x: float, h: float=np.sqrt(sys.float_info.epsilon)) -> float:
    return (f(x+h) - f(x))/h

def diff_central(f, x: float, h: float=np.cbrt(sys.float_info.epsilon)) -> float:
    return (f(x+h/2) - f(x-h/2))/h

def diff_backward(f, x: float, h: float=np.sqrt(sys.float_info.epsilon)) -> float:
    return (f(x) - f(x-h))/h

# Example usage
def func(x):
    return x**2 + np.sin(x)

x0 = 1.0
print(f"The derivative at x = {x0} is {diff_forward(func, x0)}")
```

finite_diff.py

Numerical Differentiation

Complex step method

Uses one single function evaluation after taking a step in the imaginary direction.

$$f(x+ih) = f(x) + ihf'(x) - h^2\frac{f''(x)}{2!} - ih^3\frac{f'''(x)}{3!} + \cdots$$

$$\begin{aligned}
\mathrm{Im}(f(x+ih)) &= hf'(x) - h^3\frac{f'''(x)}{3!} + \cdots \\
\Rightarrow f'(x) &= \frac{\mathrm{Im}(f(x+ih))}{h} + h^2\frac{f'''(x)}{3!} - \cdots \\
&= \frac{\mathrm{Im}(f(x+ih))}{h} + O(h^2) \text{ as } h \to 0
\end{aligned}$$

$$\mathrm{Re}(f(x+ih)) = f(x) - h^2\frac{f''(x)}{2!} + \ldots$$

$$\Rightarrow f(x) = \mathrm{Re}(f(x+ih)) + h^2\frac{f''(x)}{2!} - \cdots$$

```python
import numpy as np

def diff_complex(f, x: float, h: float=1e-20) -> float:
    return np.imag(f(x + h * 1j)) / h

# Example usage
def func(x):
    return x**2 + np.sin(x)

x0 = 1.0
print(f"The derivative at x = {x0} is {diff_complex(func, x0)}")
```

complex_diff.py

Numerical Differentiation Error Comparison

At small h, round off errors dominate, and at large h, truncation errors dominate.
Note the log transformation.

Numerical Differentiation in ML

- Approximation errors would be tolerated in a deep learning setting thanks to the well-documented error resiliency of neural network architectures (Gupta et al., 2015).
- The $O(n)$ complexity of numerical differentiation for a gradient in n dimensions is the main obstacle to its usefulness in machine learning, where n can be as large as millions or billions in state-of-the-art deep learning models (Shazeer et al., 2017).

Outline

3. Derivatives

4. Symbolic Differentiation

5. Numerical Differentiation

6. Automatic Differentiation

Automatic Differentiation

Automatic differentiation techniques are founded on the observation that any function is evaluated by performing a sequence of simple elementary operations involving just one or two arguments at a time:

- addition
- multiplication
- division
- power operation a^b
- trigonometric functions
- exponential functions
- logarithmic
- chain rule:

$$\frac{\mathrm{d}}{\mathrm{d}x} f(g(x)) = \frac{\mathrm{d}}{\mathrm{d}x} f \circ g(x) = \frac{\mathrm{d}f}{\mathrm{d}g} \frac{\mathrm{d}g}{\mathrm{d}x}$$

- Forward Accumulation is equivalent to expanding a function using the chain rule and computing the derivatives inside-out
- Requires n-passes to compute n-dimensional gradient
- Example:

$$f(a, b) = \ln(ab + \max(a, 2))$$

$$\begin{aligned}
\frac{\partial f}{\partial a} &= \frac{\partial}{\partial a} \ln(ab + \max(a, 2)) \\
&= \frac{1}{ab + \max(a, 2)} \frac{\partial}{\partial a}(ab + \max(a, 2)) \\
&= \frac{1}{ab + \max(a, 2)} \left[\frac{\partial(ab)}{\partial a} + \frac{\partial \max(a, 2)}{\partial a} \right] \\
&= \frac{1}{ab + \max(a, 2)} \left[\left(b\frac{\partial a}{\partial a} + a\frac{\partial b}{\partial a} \right) + \left((2 > a)\frac{\partial 2}{\partial a} + (2 < a)\frac{\partial a}{\partial a} \right) \right] \\
&= \frac{1}{ab + \max(a, 2)} [b + (2 < a)]
\end{aligned}$$

Automatic Differentiation

Computational graph: nodes are are operations and the edges are input-output relations. leaf nodes of a computational graph are input variables or constants, and terminal nodes are values output by the function

Forward accumulation for $f(a, b) = \ln(ab + \max(a, 2))$

Automatic Differentiation

Computational graph: nodes are are operations and the edges are input-output relations. leaf nodes of a computational graph are input variables or constants, and terminal nodes are values output by the function

Forward accumulation for $f(a, b) = \ln(ab + \max(a, 2))$

$b = 2$

$\dot{b} = 0$

$a = 3$

$\dot{a} = 1$

dual numbers

$\frac{\partial b}{\partial a} = \dot{b}$ Newton notation

Automatic Differentiation

Computational graph: nodes are are operations and the edges are input-output relations. leaf nodes of a computational graph are input variables or constants, and terminal nodes are values output by the function

Forward accumulation for $f(a, b) = \ln(ab + \max(a, 2))$

dual numbers

$\frac{\partial b}{\partial a} = \dot{b}$ Newton notation

Automatic Differentiation

Computational graph: nodes are are operations and the edges are input-output relations. leaf nodes of a computational graph are input variables or constants, and terminal nodes are values output by the function

Forward accumulation for $f(a, b) = \ln(ab + \max(a, 2))$

$b = 2$

$\dot{b} = 0$

$a = 3$

$\dot{a} = 1$

$c_1 = a \times b = 6$

$\dot{c}_1 = b\dot{a} + a\dot{b} = 2$

$c_4 = \ln c_3 = \ln 9$

$\dot{c}_4 = \dot{c}_3 / c_3 = \frac{1}{3}$

$c_3 = c_1 + c_2 = 9$

$\dot{c}_3 = \dot{c}_1 + \dot{c}_2 = 3$

$c_2 = \max(a, 2) = 3$

$$\dot{c}_2 = \begin{cases} 0 & \text{if } 2 > a \\ \dot{a} & \text{if } 2 < a \end{cases} = 1$$

dual numbers

$\frac{\partial b}{\partial a} = \dot{b}$ Newton notation

for $\frac{\partial f}{\partial b}$ set $\dot{a} = 0, \dot{b} = 1$

Dual numbers

- Dual numbers can be expressed mathematically by including the abstract quantity ϵ, where ϵ^2 is defined to be 0.

- Like a complex number, a dual number is written $a + b\epsilon$ where a and b are both real values.

- $(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon$
  $(a + b\epsilon) \times (c + d\epsilon) = (ac) + (ad + bc)\epsilon$

- by passing a dual number into any smooth function f, we get the evaluation and its derivative. We can show this using the Taylor series:

$$
\begin{aligned}
f(x) &= \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k \\
f(a+b\epsilon) &= \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(a+b\epsilon-a)^k \\
&= \sum_{k=0}^{\infty} \frac{f^{(k)}(a)b^k\epsilon^k}{k!} \\
&= f(a) + bf'(a)\epsilon + \epsilon^2 \sum_{k=2}^{\infty} \frac{f^{(k)}(a)b^k}{k!}\epsilon^{(k-2)} \\
&= f(a) + bf'(a)\epsilon
\end{aligned}
$$

Note that

$$
\begin{aligned}
(v + \dot{v}\epsilon) + (u + \dot{u}\epsilon) &= (v + u) + (\dot{v} + \dot{u})\epsilon \\
(v + \dot{v}\epsilon)(u + \dot{u}\epsilon) &= (vu) + (v\dot{u} + \dot{v}u)\epsilon \ ,
\end{aligned}
$$

satisfies the rules of differentiation

Setting:

$$
f(v + \dot{v}\epsilon) = f(v) + f'(v)\dot{v}\epsilon
$$

The chain rule follows:

$$
\begin{aligned}
f(g(v + \dot{v}\epsilon)) &= f(g(v) + g'(v)\dot{v}\epsilon) \\
&= f(g(v)) + f'(g(v))g'(v)\dot{v}\epsilon \ .
\end{aligned}
$$

Automatic Differentiation

- **Reverse accumulation** is performed in a single run using two passes $O(m \cdot \text{ops}(f))$ (forward and back) for $f : \mathbb{R}^n \to \mathbb{R}^m$
- Note: this is central to the backpropagation algorithm used to train neural networks because it needs only one pass for the n-dimensional function to find the gradient.
- implemented through two different operation overloading functions (for forward and backward)
- Many open-source software implementations are available: eg, Tensorflow

Forward implements:

$$\frac{df}{dx} = \frac{df}{dc_4}\frac{dc_4}{dx} = \frac{df}{dc_4}\left(\frac{dc_4}{dc_3}\frac{dc_3}{dx}\right) = \frac{df}{dc_4}\left(\frac{dc_4}{dc_3}\left(\frac{dc_3}{dc_2}\frac{dc_2}{dx} + \frac{dc_3}{dc_1}\frac{dc_1}{dx}\right)\right)$$

Backward implements:

$$\frac{df}{dx} = \frac{df}{dc_4}\frac{dc_4}{dx} = \left(\frac{df}{dc_3}\frac{dc_3}{dc_4}\right)\frac{dc_4}{dx} = \left(\left(\frac{df}{dc_2}\frac{dc_2}{dc_3} + \frac{df}{dc_1}\frac{dc_1}{dc_3}\right)\frac{dc_3}{dc_4}\right)\frac{dc_4}{dx}$$

Complementing each intermediate variable v_i with an **adjoint**

$$\bar{v}_i = \frac{\partial y_j}{\partial v_i}$$

which represents the sensitivity of a considered output y_j with respect to changes in v_i.

Example

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - sin(x_2)$$

Example: Forward Accumulation

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - sin(x_2)$$

Forward Primal Trace		
v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \ln v_{-1}$	$= \ln 2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

Forward Tangent (Derivative) Trace		
$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
$\dot{v}_0$	$= \dot{x}_2$	$= 0$
$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
$\boldsymbol{\dot{y}}$	$\boldsymbol{= \dot{v}_5}$	$\boldsymbol{= 5.5}$

$$O(n \cdot \mathrm{ops}(f))$$

Example: Reverse Accumulation

Forward Primal Trace

$$
\begin{aligned}
v_{-1} &= x_1 && = 2 \\
v_0 &= x_2 && = 5 \\
\hline
v_1 &= \ln v_{-1} && = \ln 2 \\
v_2 &= v_{-1} \times v_0 && = 2 \times 5 \\
v_3 &= \sin v_0 && = \sin 5 \\
v_4 &= v_1 + v_2 && = 0.693 + 10 \\
v_5 &= v_4 - v_3 && = 10.693 + 0.959 \\
\hline
y &= v_5 && = 11.652
\end{aligned}
$$

Reverse Adjoint (Derivative) Trace

$$
\begin{aligned}
\boldsymbol{\bar{x}_1} &= \boldsymbol{\bar{v}_{-1}} && = \mathbf{5.5} \\
\boldsymbol{\bar{x}_2} &= \boldsymbol{\bar{v}_0} && = \mathbf{1.716} \\
\hline
\bar{v}_{-1} &= \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}} && = \bar{v}_{-1} + \bar{v}_1 / v_{-1} && = 5.5 \\
\bar{v}_0 &= \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0} && = \bar{v}_0 + \bar{v}_2 \times v_{-1} && = 1.716 \\
\bar{v}_{-1} &= \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}} && = \bar{v}_2 \times v_0 && = 5 \\
\bar{v}_0 &= \bar{v}_3 \frac{\partial v_3}{\partial v_0} && = \bar{v}_3 \times \cos v_0 && = -0.284 \\
\bar{v}_2 &= \bar{v}_4 \frac{\partial v_4}{\partial v_2} && = \bar{v}_4 \times 1 && = 1 \\
\bar{v}_1 &= \bar{v}_4 \frac{\partial v_4}{\partial v_1} && = \bar{v}_4 \times 1 && = 1 \\
\bar{v}_3 &= \bar{v}_5 \frac{\partial v_5}{\partial v_3} && = \bar{v}_5 \times (-1) && = -1 \\
\bar{v}_4 &= \bar{v}_5 \frac{\partial v_5}{\partial v_4} && = \bar{v}_5 \times 1 && = 1 \\
\hline
\bar{v}_5 &= \bar{y} && = 1
\end{aligned}
$$

$O(m \cdot \mathrm{ops}(f))$

Summary

- Derivatives are useful in optimization because they provide information about how to change a given point in order to improve the objective function

- For multivariate functions, various derivative-based concepts are useful for directing the search for an optimum, including the gradient, the Hessian, and the directional derivative

- computation of derivatives in computer programs can be classified into four categories:

  1. manually working out derivatives and coding them (error prone and time consuming)

  2. numerical differentiation using finite difference approximations
     Complex step method can eliminate the effect of subtractive cancellation error when taking small steps

  3. symbolic differentiation using expression manipulation in computer algebra systems

  4. automatic differentiation, (aka algorithmic differentiation)
     forward and reverse accumulation on computational graphs

3. Bracketing

Bracketing

A derivative-free method to identify an interval containing a local minimum and then successively shrinking that interval

Unimodality

There exists a unique optimizer $\boldsymbol{x}^*$ such that f is monotonically decreasing for $\boldsymbol{x} \leq \boldsymbol{x}^*$ and monotonically increasing for $\boldsymbol{x} \geq \boldsymbol{x}^*$

Finding an Initial Bracket

Given a unimodal function, the global minimum is guaranteed to be inside the interval $[a, c]$ if $f(a) > f(b) < (c)$

```julia
function bracket_minimum(f, x=0; s=1e-2, k=2.0)
    a, ya = x, f(x)
    b, yb = a + s, f(a + s)
    if yb > ya
        a, b = b, a
        ya, yb = yb, ya
        s = -s
    end
    while true
        c, yc = b + s, f(b + s)
        if yc > yb
            return a < c ? (a, c) : (c, a)
        end
        a, ya, b, yb = b, yb, c, yc
        s *= k
    end
end
```

Finding an Initial Bracket

Example of `bracket_minimum` on a function

reverses direction between the first and second iteration and expands until a minimum is bracketed in the fourth iteration.

For unimodal functions, when function evaluations are limited, what is the maximal shrinckage we can achieve?

When restricted to only 2 function evaluations (queries) the most we can guarantee to shrink our interval is by just under a factor of 2.

yeilds a factor of 3.

for $\epsilon \to 0$ yields a factor of just less than 2

When restricted to only 3 function evaluations (queries) the most we can guarantee to shrink our interval is by a factor of 3.

Fibonacci Search

When restricted to n functions evaluations following the previous strategy, we are guaranteed to shrink our interval by a factor of F_{n+1}.

Fibonacci numbers: sum of previous two,
$1, 1, 2, 3, 5, 8, 13, \ldots$

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1, 2 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

The length of every interval constructed can be expressed in terms of the final interval times a Fibonacci number.
- final, smallest interval has length I_n,
- second smallest interval has length $I_{n-1} = F_3 I_n$
- third smallest interval has length $I_{n-2} = F_4 I_n$,

and so forth.

Fibonacci Search Algorithm

For a unimodal function f in the interval $[a, b]$, we want to shrink the interval within n iterations. (At each iteration we want to shrink by a factor ϕ).

$$b_{k+1} - a_{k+1} = \frac{F_{n-k+1}}{F_{n-k+2}}(b_k - a_k)$$

Therefore:

$$\begin{aligned} b_n - a_n &= \frac{F_2}{F_3}(b_{n-1} - a_{n-1}) \\ &= \frac{F_2}{F_3}\frac{F_3}{F_4}\cdots\frac{F_n}{F_{n+1}}(b_1 - a_1) \\ &= \frac{1}{F_{n+1}}(b_1 - a_1) \end{aligned}$$

Closed-form expression (Binet's formula):

$$F_n = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}},$$

$\phi = (1+\sqrt{5})/2 \approx 1.61803$ is the golden ratio.

$$\frac{F_{n+1}}{F_n} = \phi\frac{1 - s^{n+1}}{1 - s^n}, \quad s = (1-\sqrt{5})(1+\sqrt{5}) \approx -0.38$$

Suppose we have an unimodal function f in the interval $[a, b]$ and a tolerance $\epsilon = 0.01$. Let $k = 1$.

$$\frac{F_n}{F_{n+1}} = \rho_n = \frac{1 - s^n}{\phi(1 - s^{n+1})} \approx 0.6$$

1. $d_k = a_k + \frac{F_{n-k+1}}{F_{n-k+2}}(b_k - a_k)$
2. if $k \neq n - 1$:

   $$c_k = a_k + \left(1 - \frac{F_{n-k+1}}{F_{n-k+2}}\right)(b_k - a_k)$$

   otherwise: $c_k = d_k + \epsilon(a_k - d_k)$
3. if $f(c_k) < f(d_k)$: $\quad b_{k+1} = d_k, \quad d_{k+1} = c_k, \quad a_{k+1} = a_k$
   otherwise: $a_{k+1} = b_k, \quad b_{k+1} = c_k, \quad d_{k+1} = d_k$
4. $k = k + 1$, if $k = n$ go to step 5, else go to step 2
5. return (a_k, b_k) if $(a_k < b_k)$ else (b_k, a_k)

$f(c_k) < f(d_k)$

$f(c_k) > f(d_k)$

$k \neq n-1$

$$a_{k+1} + \underbrace{(1-\varrho)}_{\approx 0{,}4}(b_{k+1} - a_{k+1})$$

$$a_{k+1} + \underbrace{(1-\varrho)}_{\approx 0{,}4}\underbrace{(b_{k+1} - a_{k+1})}_{<0}$$

$k = n-1$

Golden Section Search

$$\lim_{n\to\infty} \frac{F_{n+1}}{F_n} = \lim_{n\to\infty} \frac{1}{\rho_n} = \lim_{n\to\infty} \phi \frac{1 - s^{n+1}}{1 - s^n} = \phi \approx 1.61803 \qquad \frac{1}{\phi} \approx 0.618$$

Comparison

Comparison

Quadratic Fit Search

- Leverages ability to analytically minimize quadratic functions
- Iteratively fits quadratic function to three bracketing points

Quadratic Fit Search

- If a function is locally nearly quadratic, the minimum can be found after several steps

Using Linear Algebra

- We assume that the variable y is related to $\boldsymbol{x} \in \mathbb{R}^n$ quadratically, so for some constants b_0, b_1, b_2:

$$y = b_0 + b_1 x + b_2 x^2$$

- Given the set of m points $(y_1, x_1,), \ldots, (y_3, x_3)$ in the ideal case, we have that $y_i = b_0 + b_1 x_i + b_2 x_i^2$, for all $i = 1, 2, 3$. In matrix form:

$$\begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

This can be written as $A\boldsymbol{z} = \boldsymbol{y}$ to emphasize that $\boldsymbol{z}$ are our unknowns and A and $\boldsymbol{y}$ are given.

In Python

In polynomial regression, the $m \times (n+1)$ matrix A is called a **Vandermonde matrix** (a matrix with entries $a_{ij} = x_i^{n+1-j}$, $j = 1..n+1$).
NumPy's `np.vander()` is a convenient tool for quickly constructing a Vandermonde matrix, given the values x_i, $i = 1..m$, and the number of desired columns $(n+1)$.

```
>>> print(np.vander([2, 3, 5], 2))
[[2 1]                              # [[2**1, 2**0]
 [3 1]                              #  [3**1, 3**0]
 [5 1]]                             #  [5**1, 5**0]]

>>> print(np.vander([2, 3, 5, 4], 3))
[[ 4  2  1]                         # [[2**2, 2**1, 2**0]
 [ 9  3  1]                         #  [3**2, 3**1, 3**0]
 [25  5  1]                         #  [5**2, 5**1, 5**0]
 [16  4  1]]                        #  [4**2, 4**1, 4**0]
```

In Python

```
A = np.vander(x,4)

coeff = np.linalg.solve(A,y) ## Error!! Why?

B = A.T @ A
z = np.linalg.inv(B) @ A.T @ y

coeff = np.linalg.lstsq(A, y)[0]
np.allclose(z,coeff)

f=np.poly1d(coeff)
plt.plot(x, y, 'o', label='Original data', ↪
    ↪markersize=2)
plt.plot(x, f(x), 'r', label='Fitted line')
plt.legend()
plt.show()
```

ex2.py

Shubert-Piyavskii Method

- The Shubert-Piyavskii method is guaranteed to find the global minimum of any bounded function

- but requires that the function be Lipschitz continuous

- A function is **Lipschitz continuous** if there is an upper bound on the magnitude of its derivative. A function f is Lipschitz continuous on $[a, b]$ if there exists an $\ell > 0$ such that:

$$|f(x) - f(y)| \leq \ell |x - y|, \quad \forall x, y \in [a, b]$$

y
y
y
x
x
x

Bisection Method

- **Intermediate value theorem**: If f is continuous on $[a, b]$, and there is some $y \in [f(a), f(b)]$, then there exists at least one $x \in [a, b]$, such that $f(x) = y$.
- Used in root-finding methods
- When applied to $f'(x)$, can be used to find minimum of f
- if $\text{sign}(f'(a)) \neq \text{sign}(f'(b))$, or equivalently, $f'(a)f'(b) \leq 0$ then $[a, b]$ is guaranteed to contain a zero.

Bisection method

- Cut the bracketed region $[a, b]$ in half with every iteration
- Evaluate the midpoint $(a + b)/2$
- form a new bracket from the midpoint and whichever side that continues to bracket a zero.
- Terminate after a fixed number of iterations.
- Guaranteed to converge within ϵ of x^* within $\lg_2(|b - a|/\epsilon)$

Summary

- Many optimization methods shrink a bracketing interval, including Fibonacci search, golden section search, and quadratic fit search
- The Shubert-Piyavskii method outputs a set of bracketed intervals containing the global minima, given the Lipschitz constant
- Root-finding methods like the bisection method can be used to find where the derivative of a function is zero

4. Local Descent

Preface

For multivariate functions, we have argued that:

- derivatives can have exponential growth in the resulting analytical expression
- calculating zeros might be challenging

Hence, minimizing by solving $\nabla f(\boldsymbol{x}) = 0$ may be computationally demanding.

Descent Direction Iteration

Descent Direction Methods use a local model to incrementally improve design point until some convergence criteria is met

1. Check termination conditions at $\boldsymbol{x}_k$; if not met, continue.
2. Decide **descent direction** $\boldsymbol{d}_k$ using local information
3. Decide **step size** (= magnitude of the overall step = α_k, since commonly $||\boldsymbol{d}_k||_2 = 1$)
4. Compute next design point $\boldsymbol{x}_{k+1}$

$$\boxed{\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k}$$

Line Search for Step Size

Assuming we have the search direction:

- Used to compute α
- Using the techniques discussed from previous classes, solve:

$$\text{minimize}_{\alpha} f(\boldsymbol{x} + \alpha \boldsymbol{d})$$

- Often this is computed approximately to reduce cost

Line Search: Alternatives

Step size:

- Fixed α called **learning rate** (commonly $||\boldsymbol{d}_k||_2 = 1$ not imposed)
- **Decaying step factor**

$$\alpha_k = \alpha_1 \gamma^{k-1} \qquad \text{for } \gamma \in [0, 1]$$

Decaying step factor is often required in convergence proofs

Approximate Line Search

- If function calls are expensive, rather than finding the minimum along a search direction, find a point of sufficient decrease

$$f(\mathbf{x}_{k+1}) \leq f(\mathbf{x}_k) + \beta\alpha\nabla_{\mathbf{d}_k} f(\mathbf{x}_k)$$

- $\beta \in [0, 1]$, usually $\beta = 1 \times 10^{-4}$

- Backtracking line search starts with a large step and then backs off

```
def backtracking_line_search(f, grad, x, d, alpha_0=1, p=0.5, beta=1e-4):
    y, g, alpha = f(x), grad(x), alpha_0
    while ( f(x + alpha * d) > y + beta * alpha * np.dot(g, d) ) :
        alpha *= p
    return alpha
```

Approximate Line Search: Example

Approximate Line Search

Building on backtracking line search are the Wolfe Conditions each sufficient to guarantee convergence to a local minimum.

1. First Wolfe Condition: Sufficient Decrease

$$f(\boldsymbol{x}_{k+1}) \leq f(\boldsymbol{x}_k) + \beta\alpha\nabla_{\boldsymbol{d}_k} f(\boldsymbol{x}_k)$$

2. Second Wolfe Condition: Curvature Condition

$$\nabla_{\boldsymbol{d}_k} f(\boldsymbol{x}_{k+1}) \geq \sigma\nabla_{\boldsymbol{d}_k} f(\boldsymbol{x}_k)$$

$\beta < \sigma < 1$ with
- $\sigma = 0.1$ with conjugate gradient method
- $\sigma = 0.9$ with Newton method

Approximate Line Search

The curvature condition ensures the second-order function approximations have positive curvature

$$\nabla_{\boldsymbol{d}_k} f(\boldsymbol{x}_{k+1}) \geq \sigma \nabla_{\boldsymbol{d}_k} f(\boldsymbol{x}_k)$$

Approximate Line Search

Regions satisfying the curvature condition

$$\nabla_{\boldsymbol{d}_k} f(\boldsymbol{x}_{k+1}) \geq \sigma \nabla_{\boldsymbol{d}_k} f(\boldsymbol{x}_k)$$

Approximate Line Search: Example

Consider approximate line search on $f(x_1, x_2) = x_1^2 + x_1 x_2 + x_2^2$
from $\boldsymbol{x} = [1, 2]$ in the direction $\boldsymbol{d} = [-1, -1]$, gradient at $\boldsymbol{x}$ is $\boldsymbol{g} = [4, 5]$
using a maximum step size of 10, a reduction factor of 0.5,
first Wolfe condition parameter $\beta = 1 \times 10^{-4}$, second Wolfe condition parameter $\sigma = 0.9$.

first Wolfe condition ($f(\boldsymbol{x} + \alpha \boldsymbol{d}) \leq f(\boldsymbol{x}) + \beta\alpha(\boldsymbol{g}^T \cdot \boldsymbol{d})$):

$$\begin{aligned}
\alpha = 10: &\quad f([1,2] + 10 \cdot [-1,-1]) \leq 7 + 1 \times 10^{-4} 10 [4,5]^T [-1,-1] \implies 217 \not\leq 6.991 \\
\alpha = 10 \cdot 0.5 = 5: &\quad f([1,2] + 5 \cdot [-1,-1]) \leq 7 + 1 \times 10^{-4} 5 [4,5]^T [-1,-1] \implies 37 \not\leq 6.996 \\
\alpha = 2.5: &\quad f([1,2] + 2.5 \cdot [-1,-1]) \leq 7 + 1 \times 10^{-4} 2.5 [4,5]^T [-1,-1] \implies 3.25 \leq 6.998
\end{aligned}$$

The candidate design point $\boldsymbol{x}' = \boldsymbol{x} + \alpha \boldsymbol{d} = [-1.5, -.0.5]$ is checked against the second Wolfe condition $\nabla_{\boldsymbol{d}} f(\boldsymbol{x}') \geq \sigma \nabla_{\boldsymbol{d}} f(\boldsymbol{x})$:

$$[-3.5, -2.5] \cdot [-1, -1] \geq \sigma [4, 5] \cdot [-1, -1] \implies 6 \geq -8.1$$

Approximate line search terminates with $\boldsymbol{x} = [-1.5, -0.5]$.

Approximate Line Search

Regions where the strong curvature condition is satisfied

$$|\nabla_{\boldsymbol{d}_k} f(\boldsymbol{x}_{k+1})| \leq -\sigma \nabla_{\boldsymbol{d}_k} f(\boldsymbol{x}_k)$$

Approximate Line Search

- The sufficient decrease condition with the strong curvature condition form the strong Wolfe conditions.

- Satisfying the strong Wolfe conditions requires a more complicated algorithm

  Strong backtracking line search:

  1. Bracketing Phase: tests successively larger step sizes to bracket an interval $[\alpha_{k-1}, \alpha_k]$ guaranteed to contain step lengths satisfying the Wolfe conditions.
  2. Zoom Phase: shrink the interval using bisection to find point satisfying the **strong** Wolfe conditions

Approximate Line Search

1. Bracketing Phase

An interval guaranteed to contain step lengths satisfying the Wolfe conditions is found when one of the following conditions hold:

$$f(\boldsymbol{x} + \alpha \boldsymbol{d}) \geq f(\boldsymbol{x})$$
$$f(\boldsymbol{x} + \alpha \boldsymbol{d}) > f(\boldsymbol{x}) + \beta\alpha\nabla \boldsymbol{d} f(\boldsymbol{x})$$
$$\nabla f(\boldsymbol{x} + \alpha \boldsymbol{d}) \geq 0$$

Approximate Line Search

1. Braketing Phase + zoom phase (α_5)

Trust Region Methods

- Descent methods can place too much trust in their first and second order information
- A **trust region** is the local area of the design space where the local model is believed to be reliable.
- Trust region methods, or restricted step methods, limit the step size to ensure local approximation error is minimized
- If the improvement matches the predicted value, the trust region is expanded; otherwise it is contracted

Trust Region Methods

- $\boldsymbol{x}'$ is new design point
- $\hat{f}(\boldsymbol{x}')$ is local function approximation, eg, second-order Taylor approximation
- δ is trust region radius

$$\begin{aligned} &\text{minimize}_{\boldsymbol{x}'}\ \hat{f}(\boldsymbol{x}') \\ &\text{subject to} \quad ||\boldsymbol{x} - \boldsymbol{x}'|| \leq \delta \end{aligned}$$

Constrained optimization problem.
It can be solved efficiently if $\hat{f}$ quadratic

Trust Region Methods

δ can be expanded or contracted based on performance

$$\eta = \frac{\text{actual improvement}}{\text{predicted improvement}} = \frac{f(\boldsymbol{x}) - f(\boldsymbol{x}')}{f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x}')}$$

If $\eta < \eta_1$ contract
if $\eta > \eta_2$ expand

Trust Region Methods: Example

Trust regions can be also non circular.

Trust Region Methods

Termination Conditions (commonly used together):

- Maximum Iterations: $k > k_{\max}$
- Aboslute Improvement: $f(\boldsymbol{x}_k) - f(\boldsymbol{x}_{k+1}) < \epsilon_a$
- Relative Improvement: $f(\boldsymbol{x}_k) - f(\boldsymbol{x}_{k+1}) < \epsilon_r |f(\boldsymbol{x}_k)|$
- Gradient Magnitude: $||\nabla f(\boldsymbol{x}_{k+1})|| < \epsilon_g$

Then random restart.

Summary

- Descent direction methods incrementally descend toward a local optimum.
- Univariate optimization can be applied during line search.
- Approximate line search can be used to identify appropriate descent step sizes.
- Trust region methods constrain the step to lie within a local region that expands or contracts based on predictive accuracy.
- Termination conditions for descent methods can be based on criteria such as the change in the objective function value or magnitude of the gradient.

5. First-Order Methods

Descent Direction Methods

How to select the descent direction?

- first-order methods that rely on gradient
- second-order methods that rely on Hessian information

Advantages of first order methods:

- cheap iterations: good for small and large scale optimization embedded optimization
- helpful because easy to warm restart

Limitations of first order methods:

- not hard to find challenging instances for them.
- can converge slowly.

Outline

7. Gradient Descent

8. Conjugate Descent

9. Accelerated Descents

Gradient Descent

The **steepest descent** direction at $\boldsymbol{x}_k$, at kth iteration of a **local descent iterative method**, is the one opposite to the gradient (**gradient descent**):

$$\boxed{\boldsymbol{d}_k = -\frac{\nabla f(\boldsymbol{x}_k)}{\|\nabla f(\boldsymbol{x}_k)\|}}$$

Guaranteed to lead to improvement if:

- f is smooth
- step size is sufficiently small
- $\boldsymbol{x}_k$ is not a stationary point (ie, $\nabla f(\boldsymbol{x}_k) = 0$)

Gradient Descent: Example

- Suppose we have

$$f(\boldsymbol{x}) = x_1 x_2^2$$

- The gradient is $\nabla f = [x_2^2, 2x_1x_2]$
- $x_k = [1, 2]$

$$\boldsymbol{d}_{k+1} = -\frac{\nabla f(\boldsymbol{x}_k)}{\|\nabla f(\boldsymbol{x}_k)\|} = \frac{[-4, -4]}{\sqrt{16+16}} = \left[-\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}\right]$$

Implementation

```python
class DescentMethod:
    alpha: float

class GradientDescent(DescentMethod):
    def __init__(self, f, grad, x, alpha):
        self.alpha = alpha

    def step(self, f, grad, x):
        alpha, g = self.alpha, grad(x)
        return x - alpha * g
```

Gradient Descent

<u>Theorem:</u> The next direction is orthogonal to the current direction.

<u>Proof:</u>

$$\alpha_k^* = \underset{\alpha}{\text{argmin}}\ f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$$

$$\nabla f(\boldsymbol{x}_k + \alpha_k^* \boldsymbol{d}_k) = \nabla_{\boldsymbol{d}_k} f(\boldsymbol{x}_k) = 0$$ because α_k^* is minimum

$$\nabla f(\boldsymbol{x}_k + \alpha_k^* \boldsymbol{d}_k)^T \boldsymbol{d}_k = 0$$ because directional derivative: $\nabla_{\boldsymbol{s}} f(\boldsymbol{x}) = \nabla f(\boldsymbol{x})^T \boldsymbol{s}$

$$\boldsymbol{d}_{k+1} = -\frac{\nabla f(\boldsymbol{x}_k + \alpha_k^* \boldsymbol{d}_k)}{\|\nabla f(\boldsymbol{x}_k + \alpha_k^* \boldsymbol{d}_k)\|}$$ gradient descent

$$\boldsymbol{d}_{k+1} \cdot \boldsymbol{d}_k = -\frac{\nabla f(\boldsymbol{x}_k + \alpha_k^* \boldsymbol{d}_k)}{\|\nabla f(\boldsymbol{x}_k + \alpha_k^* \boldsymbol{d}_k)\|} \cdot \boldsymbol{d}_k = 0$$ $\boldsymbol{d}_{k+1}^T \boldsymbol{d}_k = 0 \implies \boldsymbol{d}_{k+1} \perp \boldsymbol{d}_k$

$\square$

Gradient Descent: Example

2D Rosenbrock function

$$f(x, y) = (a - x)^2 + b(y - x^2)^2$$

Narrow valleys not aligned with gradient can be a problem

Outline

7. Gradient Descent

8. Conjugate Descent

9. Accelerated Descents

Conjugate Gradient

[Hestenes and Stiefel, 1950s]

For A symmetric positive definite:

$$A\boldsymbol{x} = \boldsymbol{b} \iff \underset{\boldsymbol{x}}{\text{minimize}}\ f(\boldsymbol{x}) \stackrel{def}{=} \frac{1}{2}\boldsymbol{x}^T A\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x}$$

$$\nabla f(\boldsymbol{x}) = A\boldsymbol{x} - \boldsymbol{b} \stackrel{def}{=} \boldsymbol{r}(\boldsymbol{x})$$

Conjugate Direction

Def.: A set of nonzero vectors $\{\boldsymbol{d}_0, \boldsymbol{d}_1, \ldots, \boldsymbol{d}_\ell\}$ is said to be **conjugate** with respect to the symmetric positive definite matrix A if

$$\boldsymbol{d}_i^T A \boldsymbol{d}_j = 0, \qquad \text{for all } i \neq j$$

(the vectors are linearly independent. Generally, not orthogonal.)

Theorem: Given an arbitrary $\boldsymbol{x}_0 \in \mathbb{R}^n$ and a set of conjugate vectors $\{\boldsymbol{d}_0, \boldsymbol{d}_1, \ldots, \boldsymbol{d}_{n-1}\}$ the sequence $\{\boldsymbol{x}_k\}$ generated by

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$$

where α_k is the analytical solution of $\min_\alpha f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$ given by:

$$\alpha_k = -\frac{\boldsymbol{r}_k^T \boldsymbol{d}_k}{\boldsymbol{d}_k^T A \boldsymbol{d}_k}$$

(aka, **conjugate direction algorithm**) converges to the solution $\boldsymbol{x}^*$ of the linear system and minimization problem in at most n steps.

Proof:

$$\min_{\alpha} f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$$

We can compute the derivative with respect to α:

$$\begin{aligned}\frac{\partial}{\partial \alpha} f(\boldsymbol{x} + \alpha \boldsymbol{d}) &= \frac{\partial}{\partial \alpha}(\boldsymbol{x} + \alpha \boldsymbol{d})^T A(\boldsymbol{x} + \alpha \boldsymbol{d}) - \boldsymbol{b}^T(\boldsymbol{x} + \alpha \boldsymbol{d})(+c) \\ &= \boldsymbol{d}^T A(\boldsymbol{x} + \alpha \boldsymbol{d}) - \boldsymbol{d}^T \boldsymbol{b} \\ &= \boldsymbol{d}^T(A\boldsymbol{x} - \boldsymbol{b}) + \alpha \boldsymbol{d}^T A \boldsymbol{d}\end{aligned}$$

Setting $\frac{\partial f(\boldsymbol{x}+\alpha \boldsymbol{d})}{\partial \alpha} = 0$ results in:

$$\alpha_k = -\frac{\boldsymbol{d}_k^T(A\boldsymbol{x}_k - \boldsymbol{b})}{\boldsymbol{d}_k^T A \boldsymbol{d}_k} = -\frac{\boldsymbol{d}_k^T \boldsymbol{r}(\boldsymbol{x}_k)}{\boldsymbol{d}_k^T A \boldsymbol{d}_k} \tag{1}$$

- Since the directions $\{\boldsymbol{d}_k\}$ are linearly independent, they must span the whole space $\mathbb{R}^n$. Hence, there is a set of scalars σ_k such that:

$$\boldsymbol{x}^* - \boldsymbol{x}_0 = \sigma_0 \boldsymbol{d}_0 + \sigma_1 \boldsymbol{d}_1 + \ldots + \sigma_{n-1} \boldsymbol{d}_{n-1}$$

- By premultiplying this expression by $\boldsymbol{d}_k^T A$ and using the conjugacy property, we obtain:

$$\sigma_k = \frac{\boldsymbol{d}_k^T A(\boldsymbol{x}^* - \boldsymbol{x}_0)}{\boldsymbol{d}_k^T A \boldsymbol{d}_k} \tag{2}$$

- If $\boldsymbol{x}_k$ is generated by conjugate direction algorithm, then we have

$$\boldsymbol{x}_k = \boldsymbol{x}_0 + \alpha_0 \boldsymbol{d}_0 + \alpha_1 \boldsymbol{d}_1 + \ldots + \alpha_k \boldsymbol{d}_{k-1}$$

- By premultiplying this expression by $\boldsymbol{d}_k^T A$ and using the conjugacy property, we have that

$$\boldsymbol{d}_k^T A(\boldsymbol{x}_k - \boldsymbol{x}_0) = 0$$

- and therefore

$$\begin{aligned}\boldsymbol{d}_k^T A(\boldsymbol{x}^* - \boldsymbol{x}_0) &= \boldsymbol{d}_k^T A(\boldsymbol{x}^* - \boldsymbol{x}_k + \boldsymbol{x}_k - \boldsymbol{x}_0) = \boldsymbol{d}_k^T A(\boldsymbol{x}^* - \boldsymbol{x}_k) + \boldsymbol{d}_k^T A(\boldsymbol{x}_k - \boldsymbol{x}_0) = \\ &= \boldsymbol{d}_k^T A(\boldsymbol{x}^* - \boldsymbol{x}_k) = \boldsymbol{d}_k^T (\boldsymbol{b} - A\boldsymbol{x}_k) = -\boldsymbol{d}_k^T \boldsymbol{r}_k.\end{aligned}$$

- Using this result in (2) and comparing with (1) we conclude $\alpha_k = \sigma_k$. $\square$

If the matrix A is diagonal, the contours of the function $f(\cdot)$ are ellipses whose axes are aligned with the coordinate directions

If A is not diagonal, its contours are elliptical, but they are usually not aligned with the coordinate directions.
Transform the problem to make A diagonal and minimize along the coordinate directions.

Conjugate Gradient Method

- The **conjugate gradient method** is a **conjugate direction method** with the property: In generating its set of conjugate vectors, it can compute a new vector $\boldsymbol{d}_k$ by using only the previous vector $\boldsymbol{d}_{k-1}$. Hence, little storage and computation requirements.

$$\boxed{\boldsymbol{d}_k = -\boldsymbol{r}_k + \beta_k \boldsymbol{d}_{k-1}}$$

where β_k is to be determined such that $\boldsymbol{d}_{k-1}$ and $\boldsymbol{d}_k$ must be conjugate with respect to A. By premultiplying by $\boldsymbol{d}_{k-1}^T A$ and imposing that $\boldsymbol{d}_{k-1}^T A \boldsymbol{d}_k = 0$ we find that

$$\beta_k = \frac{\boldsymbol{r}_k^T A \boldsymbol{d}_{k-1}}{\boldsymbol{d}_{k-1}^T A \boldsymbol{d}_{k-1}}$$

- Larger values of β indicate that the previous descent direction contributes more strongly.
- $\boldsymbol{d}_0$ is commonly chosen to be the steepest descent direction at $\boldsymbol{x}_0$
- Advantage with respect to steepest descent: implicitly reuses previous information about the function and thus better convergence.

Algorithm CG

Basic version:

Input: $f, \boldsymbol{x}_0$
Output: x^*
Set $\boldsymbol{r}_0 \leftarrow A\boldsymbol{x}_0 - b, \boldsymbol{d}_0 \leftarrow \boldsymbol{r}_0, k \leftarrow 0$;
while $\boldsymbol{r}_k \neq 0$ **do**

$$\alpha_k \leftarrow -\frac{\boldsymbol{d}_k^T \boldsymbol{r}(\boldsymbol{x}_k)}{\boldsymbol{d}_k^T A \boldsymbol{d}_k};$$
$$\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k;$$
$$\boldsymbol{r}_{k+1} \leftarrow A\boldsymbol{x}_{k+1} - \boldsymbol{b};$$
$$\beta_{k+1} \leftarrow \frac{\boldsymbol{r}_{k+1}^T A \boldsymbol{d}_k}{\boldsymbol{d}_k^T A \boldsymbol{d}_k};$$
$$\boldsymbol{d}_{k+1} \leftarrow -\boldsymbol{r}_{k+1} + \beta_{k+1} \boldsymbol{d}_k;$$
$$k \leftarrow k + 1;$$

Computationally improved version:

Input: $f, \boldsymbol{x}_0$
Output: x^*
Set $\boldsymbol{r}_0 \leftarrow A\boldsymbol{x}_0 - b, \boldsymbol{d}_0 \leftarrow \boldsymbol{r}_0, k \leftarrow 0$;
while $\boldsymbol{r}_k \neq 0$ **do**

$$\alpha_k \leftarrow -\frac{\boldsymbol{r}(\boldsymbol{x}_k)^T \boldsymbol{r}(\boldsymbol{x}_k)}{\boldsymbol{d}_k^T A \boldsymbol{d}_k};$$
$$\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k;$$
$$\boldsymbol{r}_{k+1} \leftarrow \boldsymbol{r}_k + \alpha_k A \boldsymbol{d}_k;$$
$$\beta_{k+1} \leftarrow \frac{\boldsymbol{r}_{k+1}^T \boldsymbol{r}_{k+1}}{\boldsymbol{r}_k^T \boldsymbol{r}_k};$$
$$\boldsymbol{d}_{k+1} \leftarrow -\boldsymbol{r}_{k+1} + \beta_{k+1} \boldsymbol{d}_k;$$
$$k \leftarrow k + 1;$$

- we never need to know the vectors $\boldsymbol{x}, \boldsymbol{r}$, and $\boldsymbol{d}$ for more than the last two iterations.
- major computational tasks: the matrix–vector product $A\boldsymbol{d}_k$, inner products $\boldsymbol{d}_k^T A \boldsymbol{d}_k$ and $\boldsymbol{r}_{k+1}^T \boldsymbol{r}_{k+1}$, and three vector sums

NonLinear Conjugate Gradient Methods

- The conjugate gradient method can be applied to nonquadratic functions as well.
- Smooth, continuous functions behave like quadratic functions close to a local minimum
- but! we do not know the value of A that best approximates f around $\boldsymbol{x}_k$. Instead, several choices for β_k tend to work well:
- Two changes:
  - α_k is computed by solving an approximate line search
  - the residual $\boldsymbol{r}$, (it was simply the gradient of f), must be replaced by the gradient of the nonlinear objective f.

NonLinear Conjugate Gradient Methods

Fletcher-Reeves Method:

Input: $f, \boldsymbol{x}_0$
Output: x^*
Evaluate $f_0 = f(\boldsymbol{x}_0), \nabla f_0 = \nabla f(\boldsymbol{x}_0)$;
Set $\boldsymbol{d}_0 \leftarrow -\nabla f_0, k \leftarrow 0$;
while $\nabla f_k \neq 0$ **do**
  Compute α_k by line search and set $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$;
  Evaluate ∇f_{k+1};
  $\beta_{k+1}^{FR} \leftarrow \frac{\nabla f_{k+1}^T \nabla f_{k+1}}{\nabla f_k^T \nabla f_k}$;
  $\boldsymbol{d}_{k+1} \leftarrow -\nabla f_{k+1} + \beta_{k+1}^{FR} \boldsymbol{d}_k$;
  $k \leftarrow k + 1$;

Polak-Ribière:

Input: $f, \boldsymbol{x}_0$
Output: x^*
Evaluate $f_0 = f(\boldsymbol{x}_0), \nabla f_0 = \nabla f(\boldsymbol{x}_0)$;
Set $\boldsymbol{d}_0 \leftarrow -\nabla f_0, k \leftarrow 0$;
while $\nabla f_k \neq 0$ **do**
  Compute α_k by line search and set $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$;
  Evaluate ∇f_{k+1};
  $\beta_{k+1}^{PR} \leftarrow \frac{\nabla f_{k+1}^T (\nabla f_{k+1} - \nabla f_k)}{\nabla f_k^T \nabla f_k}$;
  $\boldsymbol{d}_{k+1} \leftarrow -\nabla f_{k+1} + \beta_{k+1}^{FR} \boldsymbol{d}_k$;
  $k \leftarrow k + 1$;

PR with:

$$\beta_{k+1}^{+} = \max\{\beta_{k+1}^{PR}, 0\}$$

becomes PR$^+$ and guaranteed to converge (satisfies first Wolfe conditions).

The conjugate gradient method with the Polak-Ribière update. Gradient descent is shown in gray.

Outline

7. Gradient Descent

8. Conjugate Descent

9. Accelerated Descents

Accelerated Descents

- Addresses common convergence issues
- Some functions have regions with very small gradients (flat surface) where gradient descent gets stuck

Momentum

Rosenbrock function with $b = 100$

Momentum overcomes these issues by replicating the effect of physical momentum

Momentum

Momentum update equations:

$$\boldsymbol{v}_{k+1} = \beta \boldsymbol{v}_k - \alpha \nabla f(\boldsymbol{x}_k)$$
$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{v}_{k+1}$$

```
import numpy as np

class Momentum(DescentMethod):
  alpha: float # learning rate
  beta: float # momentum decay
  v: np.array # momentum

  def __init__(self, alpha, beta, f, grad, x):
    self.alpha = alpha
    self.beta = beta
    self.v = np.zeros_like(x)

  def step(self, grad, x):
    self.v = self.beta * self.v - self.alpha * grad(x↪
        ↪)
    return x + self.v
```

Nesterov Momentum

Issue of momentum: steps do not slow down enough at the bottom of a valley, overshoot.

Nesterov Momentum update equations:

$$\boldsymbol{v}_{k+1} = \beta \boldsymbol{v}_k - \alpha \nabla f(\boldsymbol{x}_k + \beta \boldsymbol{v}_k)$$

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{v}_{k+1}$$

Adagrad

- Instead of using the same learning rate for all components of $\boldsymbol{x}$,
  Adaptive Subgradient method (Adagrad) adapts the learning rate for each component of $\boldsymbol{x}$.
  For each component of $\boldsymbol{x}$, the update equation is

$$x_{i,k+1} = x_{i,k} - \frac{\alpha}{\epsilon + \sqrt{s_{i,k}}} \nabla f_i(\boldsymbol{x}_k)$$

  where

$$s_{i,k} = \sum_{j=1}^{k} (\nabla f_i(\boldsymbol{x}_j))^2$$

$$\epsilon \approx 1 \times 10^{-8}, \alpha = 0.01$$

- components of $\boldsymbol{s}$ are strictly nondecreasing, hence learning rate decreases over time

RMSProp

- Extends Adagrad to avoid monotonically decreasing learning rate by maintaining a decaying average of squared gradients

$$\hat{s}_{k+1} = \gamma \hat{s}_k + (1-\gamma)\left(\nabla f_(\boldsymbol{x}_k) \odot \nabla f(\boldsymbol{x}_k)\right), \qquad \gamma \in [0,1], \qquad \odot \text{ element-wise product}$$

Update Equation

$$\begin{aligned} x_{i,k+1} &= x_{i,k} - \frac{\alpha}{\epsilon + \sqrt{\hat{s}_{i,k}}} \nabla f_i(\boldsymbol{x}_k) \\ &= x_{i,k} - \frac{\alpha}{\epsilon + RMS(\nabla f_i(\boldsymbol{x}_k))} \nabla f_i(\boldsymbol{x}_k) \end{aligned}$$

root mean square: For n values $\{x_1, x_2, \ldots, x_n\}$

$$x_{\text{RMS}} = \sqrt{\frac{1}{n}\left({x_1}^2 + {x_2}^2 + \cdots + {x_n}^2\right)}.$$

AdaDelta

Also extends Adagrad to avoid monotonically decreasing learning rate
Modifies RMSProp to eliminate learning rate parameter entirely

$$x_{i,k+1} = x_{i,k} - \frac{RMS(\Delta x_i)}{\epsilon + RMS(\nabla f_i(\boldsymbol{x}))} \nabla f_i(\boldsymbol{x}_k)$$

Adam

- The **adaptive moment estimation method** (Adam), adapts the learning rate to each parameter.

- stores both an exponentially decaying gradient like momentum and an exponentially decaying squared gradient like RMSProp and Adadelta

- At each iteration, a sequence of values are computed

$$
\begin{aligned}
\text{Biased decaying momentum} \quad & \boldsymbol{v}_{k+1} = \beta \boldsymbol{v}_k - \alpha \nabla f(\boldsymbol{x}_k) \\
\text{Biased decaying squared gradient} \quad & \boldsymbol{s}_{k+1} = \gamma \boldsymbol{s}_k + (1-\gamma)\left(\nabla f(\boldsymbol{x}_k) \odot \nabla f(\boldsymbol{x}_k)\right) \\
\text{Corrected decaying momentum} \quad & \hat{v}_{k+1} = \boldsymbol{v}_{k+1}/(1-\gamma_{v,k}) \\
\text{Corrected decaying squared gradient} \quad & \hat{s}_{k+1} = \boldsymbol{s}_{k+1}/(1-\gamma_{s,k}) \\
\text{Next iterate} \quad & \boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha \hat{v}_{k+1}/(\epsilon + \sqrt{\hat{s}_{k+1}})
\end{aligned}
$$

- Defaults: $\alpha = 0.001, \gamma_v = 0.9, \gamma_s = 0.999, \epsilon = 1 \times 10^{-8}$

Adamax

Same as Adam, but based on the max-norm L_∞.

$$\begin{aligned} \boldsymbol{s}_{k+1} &= \gamma^\infty \boldsymbol{s}_k + (1-\gamma^\infty)\left(\|\nabla f(\boldsymbol{x}_k)\|_\infty\right) \\ &= \max\left(\gamma \boldsymbol{s}_k, \|\nabla f(\boldsymbol{x}_k)\|_\infty\right) \end{aligned}$$

Nadam

Nadam

- Nesterov-accelerated Adaptive Moment Estimation
- Adam is basically RMSProp with momentum
- We have seen that Nesterov is often more efficient
- Welcome to Nadam: Adam which uses the Nesterov momentum.

Hypergradient Descent

- Learning rate determines how sensitive the method is to the gradient signal.
- Many accelerated descent methods are highly sensitive to hyperparameters such as learning rate.
- Applying gradient descent to a hyperparameter of an underlying descent method is called hypergradient descent
- Requires computing the partial derivative of the objective function with respect to the hyperparameter

Hypergradient Descent

Summary

- Gradient descent follows the direction of steepest descent.
- The conjugate gradient method can automatically adjust to local valleys.
- Descent methods with momentum build up progress in favorable directions.
- A wide variety of accelerated descent methods use special techniques to speed up descent.
- Hypergradient descent applies gradient descent to the learning rate of an underlying descent method.

6. Second-Order Methods

Descent Direction Methods

How to select the descent direction?

- first-order methods that rely on gradient
- second-order methods that rely on Hessian information

Advantages of second order methods in descent algorithms:

- way of accelerating the iteration [Davidon mid 1950s]
- additional information that can help improve the local model for informing the selection of
  - directions and
  - step lengths

Second-Order Methods

- Locally approximate function as quadratic
- Comparison of first-order and second order approximations

Outline

10. Newton Method

11. Secant Method

12. Quasi-Newton Method

Newton's Method – Univariate

- Approximate a function using second-order Taylor series expansion
- analytically obtain the location where a quadratic approximation has a zero gradient.
- use that location as the next iteration to approach a local minimum.
- Univariate function

$$q(x) = f(x_k) + (x - x_k)f'(x_k) + \frac{(x - x_k)^2}{2} f''(x_k)$$

$\equiv$ finding roots of derivative function

$$\frac{\mathrm{d}q(x)}{\mathrm{d}x} = f'(x_k) + (x - x_k)f''(x_k) = 0$$

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

Newton's Method - Multivariate

- Multivariate function

$$f(\boldsymbol{x}) \approx q(\boldsymbol{x}) = f(\boldsymbol{x}_k) + \nabla f(\boldsymbol{x}_k)^T(\boldsymbol{x} - \boldsymbol{x}_k) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_k)^T H_k(\boldsymbol{x} - \boldsymbol{x}_k)$$

- H is the Hessian matrix
- Evaluate the gradient and set it to zero:

$$\nabla \boldsymbol{q}(x) = \nabla f(\boldsymbol{x}_k) + H(\boldsymbol{x}_k)(\boldsymbol{x} - \boldsymbol{x}_k) = 0$$

- Multivariate update rule

$$\boxed{\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - H_k^{-1}\nabla f(\boldsymbol{x}_k)}$$

- (If f is quadratic and its Hessian is positive definite, then the update converges to the global minimum in one step.)

Algorithm

Input: $\nabla f, H, \boldsymbol{x}_0, \epsilon, k_{max}$
Output: $\boldsymbol{x}^*$
Set $k = 0, \Delta = 1, \boldsymbol{x} = \boldsymbol{x}_0$;
while $\|\Delta\| > \epsilon$ and $k \leq k_{max}$ **do**
  $\Delta = H(\boldsymbol{x})^{-1}\nabla f(\boldsymbol{x})$;
  $\boldsymbol{x} = \boldsymbol{x} - \Delta$;
  $k = k + 1$;

It can be modified to only give a descent direction $\boldsymbol{d} = -H(\boldsymbol{x})^{-1}\nabla f(\boldsymbol{x})$ and leave the step size to be determined with line search.

Newton's method – Example

Minimize **Booth's function**:

$$f(x) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$$

- $\boldsymbol{x}_0 = [9, 8]$
- The gradient of Booth's function is:

$$\nabla f(x) = [10x_1 + 8x_2 - 34, 8x_1 + 10x_2 - 38]$$

- The Hessian of Booth's function is:

$$H(x) = \begin{bmatrix} 10 & 8 \\ 8 & 10 \end{bmatrix}$$

- The first iteration of Newton's method yields:

$$\begin{aligned}\boldsymbol{x}_1 &= \boldsymbol{x}_0 - H(\boldsymbol{x}_0)^{-1}\nabla f(\boldsymbol{x}_0)\\ &= \begin{bmatrix}9\\8\end{bmatrix} - \begin{bmatrix}10 & 8\\8 & 10\end{bmatrix}^{-1}\cdot\begin{bmatrix}10\cdot 9 + 8\cdot 8 - 34\\8\cdot 9 + 10\cdot 8 - 38\end{bmatrix} = \begin{bmatrix}9\\8\end{bmatrix} - \begin{bmatrix}10 & 8\\8 & 10\end{bmatrix}^{-1}\cdot\begin{bmatrix}120\\114\end{bmatrix} = \begin{bmatrix}1\\3\end{bmatrix}\end{aligned}$$

- Second iteration: The gradient at $\boldsymbol{x}_1$ is zero, so we have converged after a single iteration. The Hessian is positive definite everywhere, so $\boldsymbol{x}_1$ is the global minimum.

Newton's Method

Common causes of error in Newton's method

Newton's Method

- has quadratic convergence, meaning the difference between the minimum and the iterate is approximately squared with every iteration.

- This rate of convergence holds for Newton's method starting from x_0 within an interval $I = [x^* - \delta, x^* + \delta]$, for a root x^*, if

  1. $f''(x) \neq 0$ for all points in I,

  2. $f'''(x)$ is continuous on I, and

  3. $\frac{1}{2}\left|\frac{f'''(x_0)}{f''(x_0)}\right| < c\left|\frac{f'''(x^*)}{f''(x^*)}\right|$ for some $c < \infty$
     sufficient closeness condition, ensuring that the function is sufficiently approximated by the Taylor expansion and no overshoot.

Outline

10. Newton Method

11. Secant Method

12. Quasi-Newton Method

Secant Method – Univariate

- For univariate functions, if the second derivative is unknown, it can be approximated using the secant method

$$f''(x_k) = \frac{f'(x_k) - f'(x_{k-1})}{x_k - x_{k-1}}$$

- Update equation

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f'(x_k) - f'(x_{k-1})} f'(x_k)$$

- It requires an additional initial design point and suffers from the same problems as Newton's method and may take more iterations to converge due to approximating the second derivative.

Outline

10. Newton Method

11. Secant Method

12. Quasi-Newton Method

Quasi-Netwon Methods – Multivariate

- Automatic differentiation tools may not be applicable in many situations, and it may be much more costly to work with second derivatives in automatic differentiation software than with the gradient.
- Quasi-Newton methods, like steepest descent, require only the gradient of the objective function to be supplied at each iterate.
- By measuring the changes in gradients, they construct a model of the objective function that is good enough to produce superlinear convergence.
- The improvement over steepest descent is dramatic, especially on difficult problems.

Quasi-Netwon Methods – Multivariate

Use an approximation $Q_k \approx H^{-1}(\boldsymbol{x}_k)$

Input: $\boldsymbol{x}_0$, convergence tolerance $\epsilon > 0$, Q_0 (typically the $n \times n$ identity matrix)
Output: x^*
Set $k \leftarrow 0$;
while $\|\nabla f(\boldsymbol{x}_k)\| > \epsilon$ **do**
  Compute search direction $\boldsymbol{d}(x_k) = -Q_k \nabla f(\boldsymbol{x}_k)$;
  Set $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}(\boldsymbol{x}_k)$ where α_k is computed from a line search procedure to satisfy the Wolfe conditions;
  Define $\boldsymbol{\delta}_{k+1} \stackrel{def}{=} \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$ and $\boldsymbol{\gamma}_{k+1} \stackrel{def}{=} \nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}_k)$;
  Compute Q_{k+1};
  $k \leftarrow k + 1$;

- Davidon-Fletcher-Powell (DFP) method
- Broyden-Fletcher-Goldfarb-Shanno (BFGS) method
- Limited-memory BFGS (L-BFGS) method

Davidon-Fletcher-Powell (DFP) method

$$Q_{k+1} = Q_k - \frac{Q_k \gamma_k \gamma_k^T Q_k}{\gamma_k^T Q_k \gamma_k} + \frac{\boldsymbol{\delta}_k \boldsymbol{\delta}_k^T}{\boldsymbol{\delta}_k^T \gamma_k}$$

where all terms on the right hand side are evaluated at the same iteration k.

The update for Q in the DFP method has three properties:

- Q remains symmetric and positive definite.
- If $f(x) = \frac{1}{2}\boldsymbol{x}^T A\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{x} + c$, then $Q = A^{-1}$. Thus the DFP has the same convergence properties as the conjugate gradient method.
- For high-dimensional problems, storing and updating Q can be significant compared to other methods like the conjugate gradient method.

Broyden-Fletcher-Goldfarb-Shanno (BFGS) method

$$Q_{k+1} = Q_k - \left(\frac{\delta_k \gamma_k^T Q_k + Q_k \gamma_k \delta_k^T}{\delta_k^T \gamma_k} \right) + \left(1 + \frac{\gamma_k^T Q_k \gamma_k}{\delta_k^T \gamma_k} \right) \frac{\delta_k \delta_k^T}{\delta_k^T \gamma_k}$$

BFGS better than DFP with approximate line search but still uses an $n \times n$ dense matrix.

<u>Theorem</u>: Suppose that f is twice continuously differentiable and that the iterates generated by the BFGS algorithm converge to a minimizer $\boldsymbol{x}^*$ at which the Hessian matrix G is Lipschitz continuous Suppose also that the sequence $\|\boldsymbol{x}_k - \boldsymbol{x}^*\|$ converges to zero rapidly enough that $\sum_{k=1}^{\infty} \|\boldsymbol{x}_k - \boldsymbol{x}^*\| < \infty$. Then $\boldsymbol{x}_k$ converges to $\boldsymbol{x}^*$ at a superlinear rate (ie, faster than linear).

Limited-memory BFGS (L-BFGS) method

For **large-scale unconstrained optimization**
It stores the last m values for δ and γ rather than the full inverse Hessian ($i = 1$ oldest, $i = m$ last).
Compute $\boldsymbol{d}$ at $\boldsymbol{x}$ as $\boldsymbol{d} = -\boldsymbol{z}_m$ using:

$$\boldsymbol{q}_m = \nabla f(\boldsymbol{x}_k) \qquad \boldsymbol{q}_i = \boldsymbol{q}_{i+1} - \frac{\boldsymbol{\delta}_{i+1}^T \boldsymbol{q}_{i+1}}{\boldsymbol{\gamma}_{i+1}^T \boldsymbol{\delta}_{i+1}} \boldsymbol{\gamma}_{i+1}, \quad i = m-1, \ldots, 1$$

$$\boldsymbol{z}_0 = \frac{\boldsymbol{\delta}_m \odot \boldsymbol{\delta}_m \odot \boldsymbol{q}_m}{\boldsymbol{\gamma}_m^T \boldsymbol{\gamma}_m} \qquad \boldsymbol{z}_i = \boldsymbol{z}_{i-1} + \boldsymbol{\delta}_{i-1} \left(\frac{\boldsymbol{\delta}_{i-1}^T \boldsymbol{q}_{i-1}}{\boldsymbol{\gamma}_{i-1}^T \boldsymbol{\delta}_{i-1}} - \frac{\boldsymbol{\gamma}_{i-1}^T \boldsymbol{z}_{i-1}}{\boldsymbol{\gamma}_{i-1}^T \boldsymbol{\gamma}_{i-1}} \right), \quad i = 1, \ldots, m$$

For minimization, the inverse Hessian Q must remain positive definite.
The initial Hessian is often set to the diagonal of

$$Q_0 = \frac{\boldsymbol{\gamma}_0 \boldsymbol{\delta}_0^T}{\boldsymbol{\gamma}_0^T \boldsymbol{\gamma}_0}$$

Computing the diagonal for the above expression and substituting the result into $\boldsymbol{z}_0 = Q_0 \boldsymbol{q}_0$ results in the equation for $\boldsymbol{z}_0$.

L-BFGS

(L-BFGS two-loop recursion)

Input:

Output: $Q_k \nabla f_k = \boldsymbol{z}$

Set $\boldsymbol{q} \leftarrow \nabla f_k$;

for $i = k-1, k-2, \ldots, k-m$ **do**

  $\alpha_i \leftarrow \frac{\delta_i^T \boldsymbol{q}}{\gamma_i \cdot \boldsymbol{\delta}_i^T}$;

  $\boldsymbol{q} \leftarrow \boldsymbol{q} - \alpha_i \cdot \gamma_i$;

$\boldsymbol{z} \leftarrow Q_0 \boldsymbol{q}$;

for $i = k-m, k-m+1, \ldots, k-1$ **do**

  $\beta \leftarrow \frac{\gamma_i^T \cdot \boldsymbol{r}}{\gamma_i \cdot \boldsymbol{\delta}_i^T}$;

  $\boldsymbol{z} \leftarrow \boldsymbol{z} + \delta_i(\boldsymbol{z}_i - \beta)$;

Input:

Output: x^*

Choose starting point $\boldsymbol{x}_0$, integer $m > 0$;

$k \leftarrow 0$;

while not convergence **do**

  Set Q_0;

  Compute $\boldsymbol{d}_k \leftarrow -Q_k \nabla f_k$ from Algorithm on the left;

  Compute $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$, where α_k is chosen to satisfy the Wolfe conditions;

  if $k > m$ **then**

    Discard the vector pair $\{\boldsymbol{\delta}_{k-m}, \boldsymbol{\gamma}_{k-m}\}$ from storage;

  Compute and save $\boldsymbol{\delta}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$, $\gamma_k = \nabla f_{k+1} - \nabla f_k$;

  $k \leftarrow k+1$;

BFGS Methods - Comparison

Summary

- Incorporating second-order information in descent methods often speeds convergence.
- Newton's method is a root-finding method that leverages second-order information to quickly descend to a local minimum.
- The secant method and quasi-Newton methods approximate Newton's method when the second-order information is not directly available.
- In Python, methods implemented in the module scipy
  https://docs.scipy.org/doc/scipy/tutorial/optimize.html

7. Direct Methods

Derivative-Free Methods

- Also called **direct search methods**, zero-order, black box, pattern search
- Direct method search using function evaluations only

Cyclic Coordinate Search

- Also known as coordinate descent, or taxicab search
- Performs line search in alternating coordinate directions

$$x_{1,1} = \text{argmin}_{x_1} f(x_1, x_{2,0}, x_{3,0}, \ldots, x_{n,0})$$

$$x_{2,1} = \text{argmin}_{x_2} f(x_{1,1}, x_2, x_{3,1}, \ldots, x_{n,1})$$

Cyclic Coordinate Search

- Can be augmented to accelerate convergence
- For every full cycle starting with optimizing $\boldsymbol{x}_1$ along $[1, 0, \ldots, 0]$ and ending with $\boldsymbol{x}_{n+1}$ after optimizing along $[0, 0, \ldots, 1]$, an additional line search is conducted along the direction $\boldsymbol{x}_{n+1} - \boldsymbol{x}_1$.

Powell's Method

- Similar to Cyclic Coordinate Search, but can search in non-orthogonal directions
- Drops the oldest search direction in favor of the overall direction of progress
- It can lead the search directions to become linearly dependent and the search directions can no longer cover the full design space, and the method may not be able to find the minimum

Powell's Method

Input: $f, \boldsymbol{x}_0$
Output: $\boldsymbol{x}^*$
search directions $\boldsymbol{u}_1 = \boldsymbol{e}_1, \ldots, \boldsymbol{u}_n = \boldsymbol{e}_i$;
while not converged **do**
  for i in $\{1, \ldots, n\}$ **do**
    $\boldsymbol{x}_{i+1} \leftarrow \texttt{line_search}(f, \boldsymbol{x}_i, \boldsymbol{u}_i)$;
  for i in $\{1, \ldots, n-1\}$ **do**
    $\boldsymbol{u}_i \leftarrow \boldsymbol{u}_{i+1}$;
  $\boldsymbol{u}_n \leftarrow \boldsymbol{x}_{n+1} - \boldsymbol{x}_1$;

- search directions can become linearly dependent and no longer cover the full design space.
- peridocally reset the diections to the canonical basis.

Hooke-Jeeves

- Evaluate $f(\boldsymbol{x})$ and $f(\boldsymbol{x} \pm \alpha \boldsymbol{e}_i)$ for a given step size α in every coordinate direction from an anchoring point $\boldsymbol{x}$.
- It accepts any improvement it may find.
- If no improvements are found, it decreases the step size.
- The process repeats until the step size is sufficiently small.
- $2n$ evaluations for an n-dimensional problem

Generalized Pattern Search

- Generalization of Hooke-Jeeves method
- A pattern P can be constructed from a set of directions D about an anchoring point $\boldsymbol{x}$ with a step size α according to: $P = \{\boldsymbol{x} + \alpha \boldsymbol{d} \text{ for each } \boldsymbol{d} \in D\}$
- Searches in set of directions that **positively spans** (nonnegative linear combination) search space. (if D has full row rank and if $D\boldsymbol{x} = -D1$ with $\boldsymbol{x} \geq 0$

only positively spans the cone

only positively spans 1d space

positively spans $\mathbb{R}^2$

Generalized Pattern Search

Outline

13. Nelder-Mead Simplex Method

14. Divided Rectangles

Nelder-Mead Simplex Method

Uses simple algorithm to traverse search space using set of points defining a simplex

Initial simplex
Sort the simplex entries
Compute $\bar{\mathbf{x}}$
Compute the reflection point:
$\mathbf{x}_r = \bar{\mathbf{x}} + \alpha(\bar{\mathbf{x}} - \mathbf{x}_h)$
$y_r < y_l$?
no
$y_r \geq y_s$?
yes
$y_r \geq y_h$?
no
yes
no
yes
Replace $\mathbf{x}_h$ with $\mathbf{x}_r$
Compute the expansion point:
$\mathbf{x}_e = \bar{\mathbf{x}} + \beta(\mathbf{x}_r - \bar{\mathbf{x}})$
Compute the contraction point:
$\mathbf{x}_c = \bar{\mathbf{x}} + \gamma(\mathbf{x}_h - \bar{\mathbf{x}})$
$y_e < y_r$?
no
$y_c > y_h$?
yes
yes
no
Shrink by replacing all
$\mathbf{x}^{(i)}$ with $(\mathbf{x}^{(i)} + \mathbf{x}_\ell)/2$
Replace $\mathbf{x}_h$ with $\mathbf{x}_e$
Replace $\mathbf{x}_h$ with $\mathbf{x}_r$
Replace $\mathbf{x}_h$ with $\mathbf{x}_c$
Converged?
yes
Return best point
no

Nelder-Mead

Simplex based method [Spendley et al. (1962)]

Nelder-Mead (cont.)

Nelder-Mead (cont.)

Nelder-Mead simplex method [Nelder and Mead, 1965]:

Nelder-Mead (cont.)

Example:

Nelder-Mead (cont.)

Algorithm: Simplex search

Let $x_1, \ldots, x_{n+1}$ be vertices of a simplex

Let h: $y_h = \max_i y_i = \max f(x_i)$ and

l: $y_l = \min_i y_i = \min f(x_i)$

Let $\bar{x}$ be the centroid of points with $i \neq h$ and

$d(x_i, x_j)$ the distance between two points x_i and x_j

$(k \leftarrow 0)$

Reflect iter. k: $(k \leftarrow k+1)$ Generate the reflection x_R of x_h

Case 1 **if** $y_l \leq y_R < y_h$ then $x_h \leftarrow x_R$ and go to **Reflect**

Case 2 **else if** $y_R < y_l$ **then**

generate the expansion x_E of x_h

if $y_E < y_l$ **then** $x_h \leftarrow x_E$ and go to **Reflect**

else $x_h \leftarrow x_R$ and go to **Reflect**

Case 3 **else if** $y_R > y_i, \forall i \neq h$ **then** $x_h \leftarrow \min\{y_h, y_R\}$ and generate the contraction 1

if $y_C \leq \min\{y_h, y_R\}$ then $x_h \leftarrow x_C$ and go to **Reflect**

else contraction 2 $x_i \leftarrow (x_i + x_l)/2$

Outline

13. Nelder-Mead Simplex Method

14. Divided Rectangles

DIRECT – Divided Rectangles

- Also called DIRECT for Divided RECTangles
- Recall from Shubert-Piyavskii, a Lipshitz constant is used to provide a lower bound on the function, and a function evaluation is made where this bound is lowest

DIRECT – Divided Rectangles

- The notion of Lipschitz continuity can be extended to multiple dimensions.
  If f is Lipschitz continuous over a domain X with Lipschitz constant $\ell > 0$, then for a given design $\boldsymbol{x}_0$ and $y = f(\boldsymbol{x}_0)$, the circular cone

  $$f(\boldsymbol{x}_0) - \ell \left\| \boldsymbol{x} - \boldsymbol{x}_0 \right\|_2$$

  forms a lower bound of f

- Given m function evaluations with design points $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m\}$, we can construct a superposition of these lower bounds by taking their maximum:

  $$\max_i f(\boldsymbol{x}_i) - \ell \left\| \boldsymbol{x} - \boldsymbol{x}_i \right\|_2$$

$f(x)$
$\ell = 0.1$
$\ell = 0.2$
$\ell = 0.3$
$\ell = 0.4$
$\ell = 0.5$
$\ell = 0.6$
$\ell = 0.7$
$\ell = 0.8$
$\ell = 0.9$
$\ell = 1.0$
lower-bound minima
evaluation points
x
y

$f(x)$
$\ell = 0.1$
$\ell = 0.2$
$\ell = 0.3$
$\ell = 0.4$
$\ell = 0.5$
$\ell = 0.6$
$\ell = 0.7$
$\ell = 0.8$
$\ell = 0.9$
$\ell = 1.0$
lower-bound minima
evaluation points
x
y

interval center value $f(c)$
$(b_i - a_i)/2$
slope $= \ell$
$f(c_i)$
$f(c_i) - \ell(b_i - a_i)/2$
interval half-width $(b - a)/2$

interval center value $f(c)$
suboptimal
potentially optimal
interval half-width $(b - a)/2$

y
2
0
−2
−2
−1
0
1
2
x
0
0.5
1
1.5
2
interval half-width

Multivariate DIRECT

- intervals $\rightarrow$ hyper-rectangles
- normalizes the search space to be the unit hypercube
- divide the rectangles into thirds along the axis directions
- larger rectangles for the points with lower function evaluations
- larger rectangles are prioritized for additional splitting
- when splitting a region without equal side lengths, only the longest dimensions are split

Multivariate DIRECT

Multivariate DIRECT

Consider using DIRECT to optimize the flower function (appendix B.4) over $x_1 \in [-1, 3]$, $x_2 \in [-2, 1]$. The function is first normalized to the unit hypercube such that we optimize $x_1', x_2' \in [0, 1]$:

$$f(x_1', x_2') = \text{flower}(4x_1' - 1, 3x_2' - 2)$$

The objective function is sampled at $[0.5, 0.5]$ to obtain 0.158. We have a single interval with center $[0.5, 0.5]$ and side lengths $[1, 1]$. The interval is divided twice, first into thirds in x_1' and then the center interval is divided into thirds in x_2'.

the interval width can only take on powers of one-third, hence the interval half-width is $\left\|\frac{a-b}{2}\right\|_2 = \left\|\frac{3^{-h}}{2}\right\|_2$ where h is the depth of the rectangle

We now have five intervals:

interval	center	side lengths	vertex distance	center value
1	$[1/6, 3/6]$	$[1/3, 1]$	0.527	0.500
2	$[5/6, 3/6]$	$[1/3, 1]$	0.527	1.231
3	$[3/6, 3/6]$	$[1/3, 1/3]$	0.236	0.158
4	$[3/6, 1/6]$	$[1/3, 1/3]$	0.236	2.029
5	$[3/6, 5/6]$	$[1/3, 1/3]$	0.236	1.861

We next split on the two intervals centered at $[1/6, 3/6]$ and $[3/6, 3/6]$.

Summary

- Direct methods rely solely on the objective function and do not use derivative information.
- Cyclic coordinate search optimizes one coordinate direction at a time.
- Powell's method adapts the set of search directions based on the direction of progress.
- Hooke-Jeeves searches in each coordinate direction from the current point using a step size that is adapted over time.
- Generalized pattern search is similar to Hooke-Jeeves, but it uses fewer search directions that positively span the design space.
- The Nelder-Mead simplex method uses a simplex to search the design space, adaptively expanding and contracting the size of the simplex in response to evaluations of the objective function.
- The divided rectangles algorithm extends the Shubert-Piyavskii approach to multiple dimensions and does not require specifying a valid Lipschitz constant.

8. Stochastic Methods

Outline

15. Benchmarking

16. Stochastic Methods

Benchmarking in the COCO Platform

- Functions divided in suites.
- Functions, f_i, within suites are distinguished by their identifier $i = 1, 2, \ldots$.
- parametrized by the (input) dimension, n, and
- instance number, j. (j as an index to a continuous parameter vector setting, eg, search space translations and rotations).

$$f_i^j \equiv f[n, i, j] : \mathbb{R}^n \to \mathbb{R} \qquad \boldsymbol{x} \mapsto f_i^j(\boldsymbol{x}) = f[n, i, j](\boldsymbol{x}).$$

- Varying n or j leads to a variation of the same function i of a given suite.
- Fixing n and j of function f_i defines an **optimization problem instance** $(n, i, j) \equiv (f_i, n, j)$ that can be presented to the solver.

Why?

Varying the instance parameter j represents a natural randomization for experiments in order to:

- generate repetitions on a single function for deterministic solvers, making deterministic and non-deterministic solvers directly comparable (both are benchmarked with the same experimental setup)
- average away irrelevant aspects of the function definition
- alleviate the problem of overfitting, and
- prevent exploitation of artificial function properties

BBOB Functions

- All benchmark functions are scalable with the dimension.
- Most functions have no specific value of their optimal solution (they are randomly shifted in $\boldsymbol{x}$-space).
- All functions have an artificially chosen optimal function value (they are randomly shifted in f-space).

Runtime and Target Values

- Runtime of a solver on a problem is the hitting time condition.
- define a non-increasing quality indicator measure and prescribe a set of **target values**, t.
- target values are compared with the best so-far-seen f-value.
- For a single run, the solver run is successful on the problem instance (f_i, n, j) when the best-so-far f-value reaches the target value t.
- COCO collects hundreds of different target values from each single run.
- targets $t(i, j)$ depend on the problem instance in a way to make problems comparable
- typically, target values are set to known or estimated optimal solution plus an added **precision**
- **runtime** is the number of f-evaluations needed to solve the problem $(fi, n, j, t(i, j))$.
- only runtimes to comparable target values can be aggregated among problem instances.

Simulated Restarts

- If a solver does not hit the target t in a given single run, the run is considered to be unsuccessful.

- The runtime of this single run remains undefined but is bounded from below by the number of evaluations conducted during the run $\tau \in [T, \infty]$

- T depends on the termination condition encountered. It can be the budget of evaluations.

- For hard problem instances COCO uses **budget-based target values**:
  For any given budget, COCO selects from the finite set of recorded target values the easiest (i.e., largest) target for which the expected runtime of all solvers (ERT) exceeds the budget.

- With unsuccessful runs: draw further runs from the set of tried problem instances, uniformly at random with replacement, until find an instance, j, for which $(f_i, n, j, t(i, j))$ is solved.
  the runtime is then the sum of the overall time spend and associated to the initially unsolved problem instance.

```
print: '|'if problem.final_target_hit, ':'if restarted and '.'otherwise↪
```

Aggregation

- Aggregation is to compute a statistical summary over a set or subset of problem instances over which we assume a uniform distribution

- If we can distinguish between problems easily, for example, according to their input dimension, we can use the information to select the solver, hence not worth aggregating data

- Empirical cumulative distribution functions of runtimes (**runtime ECDFs**)
  - Absolute distributions vs Performance profiles (ECDFs of runtimes relative to the respective best solver)
  - aggregate runtimes from several targets per function (!?)

- arithmetic average, as an estimator of the expected runtime. The estimated expected runtime of the restarted solver, ERT, is often plotted against dimension to indicate scaling with dimension.
  alternatives: average of log-runtimes $\equiv$ geometric average or shifted geometric mean

bbob f1-f24, 2-D
51 targets: 100..1e-08
15 instances
best 2009
BFGS ros
my nelder
scipy nel
v2.7.1
Fraction of function,target pairs
log10(# f-evals / dimension)
0.0
0.2
0.4
0.6
0.8
1.0
0
1
2
3
4
5
6
7

Reference

Nikolaus Hansen, Anne Auger, Raymond Ros, Olaf Mersmann, Tea Tušar & Dimo Brockhoff (2021) COCO: a platform for comparing continuous optimizers in a black-box setting, Optimization Methods and Software, 36:1, 114-144, DOI: 10.1080/10556788.2020.1808977

Outline

15. Benchmarking

16. Stochastic Methods

- Employ randomness strategically to help explore design space
- Randomness can help escape local minima
- Increases chance of searching near the global minimum
- Typically rely on pseudo-random number generators to ensure repeatability
- Control over randomness and the exploration vs exploitation trade off.

Noisy Descent

- Saddle points, where the gradient is very close to zero, can cause descent methods to select step sizes that are too small to be useful

- add Gaussian noise at each descent step

$$\boldsymbol{x}_{k+1} \leftarrow x_k + \alpha \nabla f(\boldsymbol{x}_k) + \epsilon_k$$

$\epsilon_k \sim \mathcal{N}(0, \sigma_k^2)$

- $\sigma_k = \frac{1}{k}$

Stochastic Gradient Descent

- evaluates gradients using randomly chosen subsets of the training data (**batches**)
- significantly less expensive computationally than calculating the true gradient at every iteration and yields same effect as noisy gradient approximation
- helping traverse past saddle points Convergence guarantees for stochastic gradient descent require that the positive step sizes be chosen such that:

$$\sum_{k=1}^{\infty} \alpha_k = \infty \qquad \sum_{k=1}^{\infty} \alpha_k^2 < \infty$$

- ensure that the step sizes decrease and allow the method to converge, but not too quickly so as to become stuck away from a local minimum

x_2
x_1
stochastic gradient descent
steepest descent

Mesh Adaptive Direct Search

- Similar to generalized pattern search but uses random positive spanning directions
- Example: set of positive spanning sets constructed from nonzero directions $d_1, d_2 \in \{-1, 0, 1\}$.

- Construct lower triangular matrix L sampling from:

$$\{-1/\sqrt{\alpha_k}+1, -1/\sqrt{\alpha_k}+2, \ldots, 1/\sqrt{\alpha_k}-1\}$$

- permute rows and columns of L randomly to obtain a matrix D whose columns correspond to n directions that linearly span $\mathbb{R}^n$. The maximum magnitude among these directions is $1/\sqrt{\alpha_k}$

- add one additional direction $\boldsymbol{d}_{n+1} = -\sum_{i=1}^{n} \boldsymbol{d}_i$ or add n additional directions $d_{n+j} = -\boldsymbol{d}_j$

- $$\alpha_{k+1} \leftarrow \begin{cases} \alpha_k/4 & \text{if no improvement was found in this iteration} \\ \min(1, 4\alpha_k) & \text{otherwise} \end{cases}$$

- If $f(\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \alpha \boldsymbol{d}) < f(x_{k-1})$, then the queried point is $\boldsymbol{x}_{k-1} + 4\alpha \boldsymbol{d} = x_k + 3\alpha \boldsymbol{d}$

Simulated Annealing

- often used on functions with many local minima due to its ability to escape local minima.

- a candidate transition from $\boldsymbol{x}$ to $\boldsymbol{x}'$ is sampled from a transition distribution T, eg, multivariate Gaussian

$$\boldsymbol{x}' = \boldsymbol{x} + \epsilon \qquad \epsilon \sim T$$

- **Metropolis acceptance criterion:**

$$\boxed{p(\boldsymbol{x}, \boldsymbol{x}') = \begin{cases} 1 & \text{if } \Delta \leq 0 \\ e^{-\frac{\Delta}{t_k}} & \text{if } \Delta > 0 \end{cases}}$$

where $\Delta = f(\boldsymbol{x}') - f(\boldsymbol{x})$ is the change in the objective function value and t_k is the **temperature** at iteration k.

Annealing Plan

- a logarithmic annealing schedule

$$t_k = t_0 \frac{\ln(2)}{\ln(k+1)}$$

  guaranteed to asymptotically reach the global optimum under certain conditions, but it can be slow in practice.
- exponential annealing schedule, more common, uses a simple decay factor:

$$t_{k+1} = \gamma t_k$$

- fast annealing

$$t_k = \frac{t_0}{k}$$

Simulated Annealing

- Corana et al 1987 introduced variable step-size $\boldsymbol{v}$ (separate directional components)
- cycle of random moves, one in each direction

$$\boldsymbol{x}' = \boldsymbol{x} + r v_i e_i$$

where r is randomly sampled from $\{-1, 1\}$

- after n_s cycles, step size is adjusted according to

$$v_i = \begin{cases} v_i \left(1 + c_i \frac{a_i/n_s - 0.6}{0.4}\right) & \text{if } a_i > 0.6 n_s \\ v_i \left(1 + c_i \frac{0.4 - a_i/n_s}{0.4}\right)^{-1} & \text{if } a_i < 0.4 n_s \\ v_i & \text{otherwise} \end{cases}$$

$\boldsymbol{a}$: accepted steps in each direction; $\boldsymbol{c}$: typically 2.

regulates the ratio of accepted-to-rejected points to about 50%.

Simulated Annealing

- Temperature reduction occurs every n_t step adjustments, which is every $n_s \cdot n_t$ cycles
- termination when the temperature sinked low and no improvement expected or when no movement more than ϵ in last n_ϵ iterations

Cross-Entropy Method

- Maintains explicit probability distribution over design space often called a **proposal distribution**
- Requires choosing a family of parameterized distributions
- At each iteration, a set of design points are **conditionally independently** sampled from the proposal distribution; these are evaluated and ranked
- The best-performing subset of samples, called **elite samples**, are retained
- The proposal distribution parameters are then **updated** based on the elite samples, and the next iteration begins

Cross-Entropy Method

- Cross-entropy is a measure of divergence between two probability distributions p and q (related to Kullback-Leibler divergence)

- Here we measure cross-entropy in a case where one distribution (the one of optimal solutions) is unknown.

- A model is created and then its cross-entropy is measured on the elite set to assess how accurate the model is in predicting this set.

- Let q be the true distribution of the optimal solutions, and p the distribution of solutions as predicted by the model. Since the true distribution is unknown, cross-entropy cannot be directly calculated. Instead, an estimate of cross-entropy is:

$$H(T, p) = -\sum_{i=1}^{N} \frac{1}{N} \log_2 p(x_i)$$

where N is the size of the elite set, and $p(x)$ is the probability of solution x estimated from the training set T.

Cross-Entropy Method

cross-entropy $\equiv$ Maximum likelihood estimation

- A widely used frequentist estimator is maximum likelihood, in which $\boldsymbol{\theta}$ is set to the value that maximizes the **likelihood function** $p(\boldsymbol{x} \mid \boldsymbol{\theta})$.
- This corresponds to choosing the value of $\boldsymbol{\theta}$ for which the probability of the observed data set is maximized.
- In the machine learning literature, the **negative log of the likelihood function** is called an error function. Because the negative logarithm is a monotonically decreasing function, maximizing the likelihood is equivalent to minimizing the error.
- Suppose our data set consists of N data points $\boldsymbol{x} = \{x_1, \ldots, x_N\}$:

$$\mathcal{L}(\boldsymbol{x} \mid \boldsymbol{\theta}) = p(\boldsymbol{x} \mid \boldsymbol{\theta}) = \prod_{i=1}^{N} p(x_i \mid \boldsymbol{\theta}) \qquad \text{likelihood}$$

$$\mathcal{E}(\boldsymbol{x} \mid \theta) = -\log \mathcal{L}(\boldsymbol{x}) \qquad \text{error function}$$

Cross-Entropy Method

- $$\min_{\boldsymbol{\theta}} \mathcal{E}(\boldsymbol{x} \mid \theta) = \min_{\boldsymbol{\theta}} \left(-\log \mathcal{L}(\boldsymbol{x})\right) = -\max_{\boldsymbol{\theta}} \log \mathcal{L}(\boldsymbol{x})$$ maximum log-likelihood

$$\min \left(-\sum_{i=1}^{N} \log p(x_i \mid \boldsymbol{\theta}) \right)$$

- hence minimizing the negative of the log-likelhoood is equivalent to minimizing the entropy

Multivariate normal distribution

Probability density function

- 1-dimensional:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

- 2-dimensional:

$$f(x,y) = \frac{1}{2\pi\sigma_X\sigma_Y\sqrt{1-\rho^2}} \exp\left(-\frac{1}{2\left[1-\rho^2\right]}\left[\left(\frac{x-\mu_X}{\sigma_X}\right)^2 - 2\rho\left(\frac{x-\mu_X}{\sigma_X}\right)\left(\frac{y-\mu_Y}{\sigma_Y}\right) + \left(\frac{y-\mu_Y}{\sigma_Y}\right)^2\right]\right)$$

where ρ is the correlation between X and Y and where $\sigma_X > 0$ and $\sigma_Y > 0$. In this case,

$$\boldsymbol{\mu} = \begin{pmatrix}\mu_X \\ \mu_Y\end{pmatrix}, \quad \boldsymbol{\Sigma} = \begin{pmatrix}\sigma_X^2 & \rho\sigma_X\sigma_Y \\ \rho\sigma_X\sigma_Y & \sigma_Y^2\end{pmatrix}.$$

- d dimensional:

$$f_{\mathbf{X}}(x_1,\ldots,x_d) = \frac{\exp\left(-\frac{1}{2}\left(\boldsymbol{x}-\boldsymbol{\mu}\right)^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\left(\boldsymbol{x}-\boldsymbol{\mu}\right)\right)}{\sqrt{(2\pi)^k|\boldsymbol{\Sigma}|}}$$

with symmetric covariance matrix $\boldsymbol{\Sigma}$ positive definite.

(Disadvantage: it is unimodal)

Natural Evolution Strategies

- Similar to cross-entropy method, except instead of parameterizing distribution based on elite samples, it is optimized using **gradient descent**

- The aim is to minimize the expectation

$$E_{\boldsymbol{x}\sim p(\cdot|\boldsymbol{\theta})}[f(\boldsymbol{x})].$$

- The distribution parameter gradient is estimated from the set of function evaluations

Input: $f, \boldsymbol{\theta}, k_{MAX}, N = 100, \alpha = 0.01$
Output: $\boldsymbol{\theta}$
for k in $1, \ldots, k_{MAX}$ **do**
  Let $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ be a conditionally independent sample of size N from $p(\boldsymbol{\theta})$;
  $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha \frac{1}{N} \sum_{i=1}^{N} f(\boldsymbol{x}_i) \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{x}_i, \boldsymbol{\theta}_k)$;

Natural Evolution Strategies

$$
\begin{aligned}
\nabla_{\boldsymbol{\theta}} \, \mathbb{E}_{\mathbf{x} \sim p(\cdot \mid \boldsymbol{\theta})}[f(\mathbf{x})] &= \int \nabla_{\boldsymbol{\theta}} \, p(\mathbf{x} \mid \boldsymbol{\theta}) f(\mathbf{x}) \, d\mathbf{x} \\
&= \int \frac{p(\mathbf{x} \mid \boldsymbol{\theta})}{p(\mathbf{x} \mid \boldsymbol{\theta})} \nabla_{\boldsymbol{\theta}} \, p(\mathbf{x} \mid \boldsymbol{\theta}) f(\mathbf{x}) \, d\mathbf{x} \\
&= \int p(\mathbf{x} \mid \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x} \mid \boldsymbol{\theta}) f(\mathbf{x}) \, d\mathbf{x} \\
&= \mathbb{E}_{\mathbf{x} \sim p(\cdot \mid \boldsymbol{\theta})}[f(\mathbf{x}) \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x} \mid \boldsymbol{\theta})] \\
&\approx \frac{1}{m} \sum_{i=1}^{m} f(\mathbf{x}^{(i)}) \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}^{(i)} \mid \boldsymbol{\theta})
\end{aligned}
$$

Natural Evolution Strategies

The multivariate normal distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$ is a popular distribution family due to having analytic solutions. The likelihood in d dimensions has the form

$$p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = (2\pi)^{-\frac{d}{2}} |\boldsymbol{\Sigma}|^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})\right)$$

where $|\boldsymbol{\Sigma}|$ is the determinant of $\boldsymbol{\Sigma}$. The log likelihood is

$$\log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{d}{2} \log(2\pi) - \frac{1}{2} \log |\boldsymbol{\Sigma}| - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})$$

The parameters can be updated using their log likelihood gradients:

$$\nabla_{(\boldsymbol{\mu})} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})$$

$$\nabla_{(\boldsymbol{\Sigma})} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{2}\boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} - \frac{1}{2}\boldsymbol{\Sigma}^{-1}$$

The term $\nabla_{(\boldsymbol{\Sigma})}$ contains the partial derivative of each entry of $\boldsymbol{\Sigma}$ with respect to the log likelihood.

Natural Evolution Strategies

Directly updating $\mathbf{\Sigma}$ may not result in a positive definite matrix, as is required for covariance matrices. One solution is to represent $\mathbf{\Sigma}$ as a product $\mathbf{A}^\top \mathbf{A}$, which guarantees that $\mathbf{\Sigma}$ remains positive semidefinite, and then update $\mathbf{A}$ instead. Replacing $\mathbf{\Sigma}$ by $\mathbf{A}^\top \mathbf{A}$ and taking the gradient with respect to $\mathbf{A}$ yields:

$$\nabla_{(\mathbf{A})} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \mathbf{A}) = \mathbf{A}\left[\nabla_{(\mathbf{\Sigma})} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \mathbf{\Sigma}) + \nabla_{(\mathbf{\Sigma})} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \mathbf{\Sigma})^\top\right]$$

Natural Evolution Strategies

Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES)

- Same approach as natural evolution strategy and cross entropy method, but the proposal distribution is a multivariate Gaussian parameterized by a covariance matrix.
- At every iteration, m designs are sampled from the multivariate Gaussian:

  $$\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu}, \sigma^2 \Sigma)$$

  parameters: mean vector $\boldsymbol{\mu}$, a covariance matrix Σ, and an additional step-size scalar σ.
- The covariance matrix only increases or decreases in a single direction with every iteration, whereas σ is adapted to control the overall spread of the distribution.
- Design points are sorted $f(x^{(1)}) \leq f(x^{(2)}) \leq \ldots \leq f(x^{(m)})$.
- A new mean vector $\boldsymbol{\mu}_{k+1}$ is formed using a weighted average of the first m_e-elite sampled designs:

  $$\boldsymbol{\mu}(k+1) \leftarrow \sum_{i=1}^{m_e} w_i x^{(i)}$$

CMA-ES

- the first m_e elite weights sum to 1, and all the weights approximately sum to 0 and are ordered largest to smallest

- positive and negative weigths, more aggressive shift

- The step size σ is updated using a cumulative vector $\boldsymbol{p}_1$ that tracks steps over time: Comparing the length of $\boldsymbol{p}_1$ to its expected length under random selection provides the mechanism by which σ is increased or decreased.

- covariance matrix is updated using a cumulative vector $\boldsymbol{p}_2$ and adjusted weights;
  the update consists of three components: the previous covariance matrix Σ_k, a rank-one update, and a rank-$\boldsymbol{\mu}$ update
  Rank-one updates using the cumulation vector allow for correlations between consecutive steps to be exploited

- covariance estimated around original mean $\boldsymbol{\mu}_k$ (cross-entropy did it around new mean $\boldsymbol{\mu}_{k+1}$)

Summary

- Stochastic methods employ random numbers during the optimization process
- Simulated annealing uses a temperature that controls random exploration and which is reduced over time to converge on a local minimum
- The cross-entropy method and evolution strategies maintain proposal distributions from which they sample in order to inform updates
- Natural evolution strategies uses gradient descent with respect to the log likelihood to update its proposal distribution
- Covariance matrix adaptation is a robust and sample-efficient optimizer that maintains a multivariate Gaussian proposal distribution with a full covariance matrix

9. Population-based Methods

Population Methods

- Instead of optimizing a single design point, population methods optimize a collection of **individuals**
- A large number of individuals prevents algorithm from being stuck in a local minimum
- Useful information can be shared between individuals
- Stochastic in nature
- Easy to parallelize

Initialization

- Population methods begin with an initial population
- Common initializations are uniform, normal distribution, and Cauchy distribution
- But also space filling designs (later)

Genetic Algorithms

- Inspired by biological evolution where the fittest individuals pass their genetic information to the next generation
- Individuals are interpreted as **chromosomes**
- The fittest individuals are determined by **selection**
- The next generation is formed by selecting the fittest individuals and performing **crossover** and **mutation**

Genetic Algorithms: Chromosomes

- Simplest representation is the binary string chromosome

- Chromosomes are more commonly represented as real-valued chromosomes which are simply real-valued vectors
- Typically initialized randomly

Genetic Algorithms: Selection

- Determining which individuals pass their genetic information on to the next generation choosing chromosomes to use as parents for the next generation
- Truncation selection: truncate the lowest performers
- Tournament selection: selects fittest out of k randomly chosen individuals
- Roulette Wheel selection: individuals are chosen with probability proportional to their fitness

Genetic Algorithms: Selection

Truncation Selection
Tournament Selection
Roulette Wheel Selection

Genetic Algorithms: Crossover

- Combines the chromosomes of the parents to form children
- Single-point crossover: swap occurs after single crossover point

- Two-point crossover: two crossover points

- Uniform crossover: each bit has 50% chance of crossover

- real values are linearly interpolated between the parents' values x_a and x_b:

Genetic Algorithms: Mutation

- Mutation supports exploration of new areas of design space
- Each bit or real-valued element has a probability of being flipped or modified by noise
- The probability of an element mutating is called **mutation rate**

Genetic Algorithms

Genetic algorithm with truncation selection, single point crossover, and Gaussian mutation applied to Michalewicz function

Differential Evolution

Improves each individual $\boldsymbol{x}$ by recombining other individuals according to a simple formula

1. Choose three random, distinct individuals $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$
2. Construct interim design $\boldsymbol{z} = \boldsymbol{a} + w(\boldsymbol{b} - \boldsymbol{c})$
3. Choose a random dimension to optimize in
4. Construct candidate $\boldsymbol{x}'$ via binary crossover of $\boldsymbol{x}'$ and $\boldsymbol{z}$

$$x_i' = \begin{cases} z_i & \text{if } i = j \text{ or with probability } p \\ x_i & \text{otherwise} \end{cases}$$

5. Insert better design between $\boldsymbol{x}$ and $\boldsymbol{x}'$ into next generation

x_2
x_2
x_1
x_1
x_1
x_1

Particle Swarm Optimization

- Each individual, or particle, tracks the following
  - Current position
  - Current velocity
  - Best position seen so far by the particle
  - Best position seen so far by any particle
- At each iteration, these factors produce force and momentum effects to determine each particle's movement

$$x_i \leftarrow x_i + v_i$$

$$v_i \leftarrow wv_i + c_1 r_1 \left(x_i^{best} - x_i\right) + c_2 r_2 \left(x_i^{best} - x_i\right)$$

$\boldsymbol{x}_{best}$ is the best location found so far over all particles; w, c_1, and c_2 are parameters; and r_1 and r_2 are random numbers drawn from $U(0, 1)$

Particle Swarm Optimization

Firefly Algorithm

- Inspired by the way fireflies flash their lights to attract mates
- Attractiveness is determined by low function value
- At each iteration, fireflies move toward the most attractive lights
- Random noise is added to increase exploration

Cuckoo Search

- ...

...

The Evolutionary Computation Bestiary
http://fcampelo.github.io/EC-Bestiary/

Hybrid Methods

- Generally, population methods are good at finding the best regions in design space, but do not perform as well as descent methods near the minimizer
- Hybrid methods try to leverage the strength of both methods
- Two hybrid approaches
  Lamarckian learning
  Baldwinian learning

Hybrid Methods

- Lamarckian learning
  Performs regular descent method update on each individual

- Baldwinian learning
  Uses value of descent method update to augment the objective value of each design point

Summary

- Population methods use a collection of individuals in the design space to guide progression toward an optimum
- Genetic algorithms leverage selection, crossover, and mutations to produce better subsequent generations
- Differential evolution, particle swarm optimization, the firefly algorithm, and cuckoo search include rules and mechanisms for attracting design points to the best individuals in the population while maintaining suitable state space exploration
- Population methods can be extended with local search approaches to improve convergence

10. Machine Learning Applications

Introduction

Large-scale machine learning represents a distinctive setting in which traditional nonlinear optimization techniques typically falter

- How do optimization problems arise in machine learning applications and what makes them challenging?
- What have been the most successful optimization methods for large-scale machine learning and why?
- What recent advances have been made in the design of algorithms and what are open questions in this research area?

Two Case Studies

- Logistic regression or support vector machines
  convex optimization problems

- Deep neural networks
  highly nonlinear and nonconvex problems

Text Classification via Convex Optimization

Task: determining whether a text document is one that discusses politics.

- set of examples $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$, where for each $i \in \{1, \ldots, n\}$
  $\boldsymbol{x}_i$ represents the features of a text document (e.g., the words it includes)
  y_i is a label indicating whether the document belongs ($y_i = 1$) or not ($y_i = -1$) to a particular class.

- h prediction function

- measure performance: count how often the program prediction $h(\boldsymbol{x}_i)$ differs from the correct prediction y_i.

- minimize **empirical risk misclassification**

$$R_n(h) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}[h(\boldsymbol{x}_i) \neq y_i], \qquad \text{where} \quad \mathbb{I}[A] = \begin{cases} 1 & \text{if } A \text{ is true,} \\ 0 & \text{otherwise} \end{cases}$$

Text Classification via Convex Optimization

Choosing between prediction functions belonging to a given class by comparing them using cross-validation procedures that involve splitting the examples into three disjoint subsets:

- a training set, optimizing the choice of h by minimizing R_n
- a validation set, generalized performance of each of these remaining candidates is then estimated using the validation set, the best performing of which is chosen as the selected function.
- a testing set, only used to estimate the generalized performance of this selected function

Formalization

- feature vector $\boldsymbol{x} \in \mathbb{R}^d$ whose components are associated with a prescribed set of vocabulary words; $\|\boldsymbol{x}\| = 1$

- $h(\boldsymbol{x}; \boldsymbol{w}, \tau) = \boldsymbol{w}^T \boldsymbol{x} - \tau$, $\boldsymbol{w} \in \mathbb{R}^d$ and $\tau \in \mathbb{R}^d$

- $\text{sign}(h(\boldsymbol{x}; \boldsymbol{w}, \tau)$ discountinuous

- continuous approximation through a loss function that measures a cost for predicting h when the true label is y;
  e.g., one may choose a **log-loss function** of the form

$$L(h, y) = \log(1 + \exp -hy).$$

$$\min_{(\boldsymbol{w},\tau)\in\mathbb{R}^d\times\mathbb{R}} L(h(\boldsymbol{x}_i; \boldsymbol{w}, \tau), y_i) + \lambda \|\boldsymbol{w}\|_2^2$$

  solve for various λ and choose on the validation set

Deep Neural Networks

Deep Neural Networks: represent hypotheses as computation graphs with tunable weights and compute the gradient of the loss function with respect to those weights in order to fit the training data.

https://playground.tensorflow.org/

Perceptual Tasks via Deep Neural Networks

- Prediction function h whose value is computed by applying successive transformations to a given input vector $x_i \in \mathbb{R}^{d_0}$.
- These transformations are made in layers. A canonical fully connected layer performs the computation

$$\boldsymbol{x}_i^{(j)} = s(W_j \boldsymbol{x}_i^{(j-1)} + b_j) \in \mathbb{R}^{d_j}$$

- where $\boldsymbol{x}_i^{(0)} = \boldsymbol{x}_i$, the matrix $W_j \in \mathbb{R}^{d_j \times d_{j-1}}$ and vector $b_j \in \mathbb{R}^{d_j}$ contain the jth layer parameters, and s is a component-wise nonlinear activation function
- $s(x) = 1/(1 + \exp(-x))$ and the hinge function $s(x) = \max(0, x)$ (often called a rectified linear unit (ReLU) in this context)
- $\boldsymbol{x}_i^{(J)}$ leads to the prediction function value $h(\boldsymbol{x}_i; \boldsymbol{w})$, $\boldsymbol{w} = \{(W_1, b_1), \ldots, (W_J, b_J)\}$.
- leads to highliy non-linear and non-convex:

$$\min_{\boldsymbol{w} \in \mathbb{R}^d} \frac{1}{N} \sum_{i=1}^{n} L(h(\boldsymbol{x}_i; \boldsymbol{w}), y_i)$$

- The gradient with respect to $\boldsymbol{w}$ is made of simple expressions that can be computed by passing information back through the network from the output units.
- the gradient computations for any feedforward computation graph have the same structure as the underlying computation graph.
- gradients can be computed by the chain rule and the algorithmic method of automatic differentiation
- back-propagation in deep learning is simply an application of **reverse mode differentiation**, which applies the chain rule "from the outside in"

Speech recognition

- A contemporary fully connected neural network for speech recognition typically has five to seven layers. This amounts to tens of millions of parameters to be optimized,
- the training may require up to thousands of hours of speech data (representing hundreds of millions of training examples) and weeks of computation on a supercomputer

Convolutional neural networks

Convolutional neural networks (CNNs) have proved to be very effective for computer vision and signal processing tasks

ImageNet Large Scale Visual Recognition Competition (ILSVRC) with five convolutional layers and three fully connected layers

Image Recognition

- input $\boldsymbol{x}_i^{(j-1)}$ is intepreted as a multichannel image of 224×224 pixels.
- convolutional layers, wherein the parameter matrix W_j is a circulant matrix
- product $W_j \boldsymbol{x}_i^{(j-1)}$ computes the convolution of the image by a trainable filter
- activation functions are piecewise linear functions and can perform more complex operations that may be interpreted as image rectification, contrast normalization, or subsampling.
- output scores represent the odds that the image belongs to each of 1,000 categories.
- 60 million parameters
- training on a few million labeled images takes a few days on a dual GPU workstation.

Fundamentals

- Joint probability distribution function $P(x, y)$ that simultaneously represents the distribution $P(x)$ of inputs as well as the conditional probability $P(y \mid x)$ of the label y being appropriate for an input x.

- One should seek to find h that yields a small expected risk of misclassification over all possible inputs, i.e., an h that minimizes

  $$R(h) = P[h(\boldsymbol{x}) \neq y] = E[\mathbb{I}[h(\boldsymbol{x}) \neq y]],$$

  which is **variational** since we are optimizing over a set of functions (the h), and is **stochastic** since the objective function involves an expectation.

- without explicit knowledge of P the only tractable option is to construct a surrogate problem that relies solely on the examples $(\boldsymbol{x}_i, y_i)_{i=1}^n$: minimize the empirical risk

Tasks:

- how to choose the parameterized family of prediction functions $\mathcal{H}$ and

- how to determine (and find) the particular prediction function $h \in \mathcal{H}$ that is optimal.

Choice of Prediction Function

1. $\mathcal{H}$ should contain prediction functions that are able to achieve a low empirical risk over the training set, so as to avoid bias or underfitting the data. (rich family of functions or by using a priori knowledge to select a well-targeted family)

2. the gap between expected risk and empirical risk, namely, $R(h) - R_n(h)$, should be small over all $h \in \mathcal{H}$. (increases with rich family of functions)

3. $\mathcal{H}$ should be effciently solvable in the corresponding optimization problem (the richer the family of functions and/or the larger training set the more complex the problem becomes)

Choice of Prediction Function

Uniform laws of large numbers and the Hoeffding inequality gurantee that with probability at least $1 - \eta$

$$\sup_{h \in \mathcal{H}} |R(h) - R_n(h)| \leq \mathcal{O}\left(\sqrt{\frac{1}{2n}\log\left(\frac{2}{\eta}\right) + \frac{d_{\mathcal{H}}}{n}\log\left(\frac{n}{d_{\mathcal{H}}}\right)}\right)$$

- $d_{\mathcal{H}}$ Vapnik-Chervonenkis (VC) dimension (measure of capacity of separating points)
- not the same as numer of parameters

- for a fixed $d_{\mathcal{H}}$, uniform convergence is obtained by increasing the number of training points n.
- for a fixed n, the gap can widen for larger $d_{\mathcal{H}}$.

In practice it is typically easier to estimate with cross-validation experiments.

Structural Risk Minimization

- Rather than choosing a generic family of prediction functions one chooses a structure, i.e., a collection of nested function families.
- structure can be formed as a collection of subsets of a given family $\mathcal{H}$: given a preference function Ω, choose various values of a hyperparameter C, according to each of which one obtains the subset $\mathcal{H}_C \stackrel{def}{=} \{h \in \mathcal{H} : \omega(h) \leq C\}$. ($C$ is, eg, degree of a polynomial model function, dimension of an inner layer of a DNN)

$$\min R_n(h)$$
$$\text{subject to} \quad \Omega(h) \leq C$$

Regularized empirical risk

$$R_n(h) + \lambda\Omega(h)$$

validation set is then used to estimate the expected risk corresponding to each C and to choose one.

Structural Risk Minimization

Another approach:

- employ an algorithm for minimizing R_n, but terminate the algorithm early, i.e., before an actual minimizer of R_n is found.
  The hyperparameter is played by the training time allowed

- often essential due to computational budget limitations.

Formal Optimization Problem Statements

- We do not consider a variational optimization problem over $\mathcal{H}$,
- instead we assume that the prediction function h has a fixed form and is parameterized by a real vector $\boldsymbol{w} \in \mathbb{R}^d$
- for some given $h(\cdot;\cdot) : \mathbb{R}^{d_x} \times \mathbb{R}^d \to \mathbb{R}^{d_y}$, we consider the family of prediction functions $\mathcal{H} \stackrel{def}{=} \{h(\cdot;\boldsymbol{w}) : \boldsymbol{w} \in \mathbb{R}^d\}$
- aim to find $h \in \mathcal{H}$ that minimizes a given **loss function** $L : \mathbb{R}^{d_x} \times \mathbb{R}^d \to \mathbb{R}^{d_y}$, $L(h(x;\boldsymbol{w}), y)$
- Ideally, the expected loss is defined over **any** input-output pair. Assuming probability distribution $P(\boldsymbol{x}, y)$ represents the true input-output relationship:

$$R(w) = \int_{\mathbb{R}^{d_x} \times \mathbb{R}^{d_y}} L(h(x;w), y) dP(x,y) = E[L(h(x;w), y)] \qquad \text{Expected Risk}$$

Formal Optimization Problem Statements

- In practice, one seeks the solution of a problem that involves an estimate of the expected risk R.

- In supervised learning, we have access (either all-at-once or incrementally) to a set of $n \in \mathbb{N}$ independently drawn input-output samples $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n \subseteq \mathbb{R}^{d_x} \times \mathbb{R}^{d_y}$, with which we define the **empirical risk function** $R_n : \mathbb{R}^d \to \mathbb{R}$ by

$$R_n(\boldsymbol{w}) \stackrel{def}{=} \frac{1}{n} \sum_{i=1}^n L(h(\boldsymbol{x}_i; \boldsymbol{w}), y_i) \qquad \text{Empirical Risk}$$

(note that before we used misclassification error while now L.)

11. Convergence Analysis of Stochastic Gradient

Simplified Notation

Let ξ be a random seed or the realization of a single (or a set of) sample $(\boldsymbol{x}, y)$.
For a given $(\boldsymbol{w}, \xi)$ let $f(\boldsymbol{w}; \xi)$ be the composition of the loss function L and the prediction function h
Then:

$$R(\boldsymbol{w}) = \mathbb{E}_{\xi}[f(\boldsymbol{w}; \xi)] \qquad \text{Expected Risk}$$

Let $\{\xi_{[i]}\}_{i=1}^{n}$ be realizations of ξ corresponding to $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$ and $f_i(\boldsymbol{w}) \stackrel{def}{=} f(\boldsymbol{w}; \xi_{[i]})$
Then:

$$R_n(\boldsymbol{w}) \stackrel{def}{=} \frac{1}{n} \sum_{i=1}^{n} f_i(\boldsymbol{w}) \qquad \text{Empirical Risk}$$

Stochastic vs Batch Optimization Methods

Reduction to minimizing R_n, with $\boldsymbol{w}_0 \in \mathbb{R}^d$ given (deterministic problem)

Stochastic Approach: Stochastic Gradient (Robbins and Monro, 1951)

$$\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k - \alpha_k \nabla f_{i_k}(\boldsymbol{w}_k)$$

i_k is chosen randomly from $\{1, \ldots, n\}$, $\alpha_k > 0$.

- very cheap iteration only on one sample.
- $\{\boldsymbol{w}_k\}$ is a stochastic process determined by the random sequence $\{i_k\}$.
- the direction might not always be a descent but if it is a descent direction in **expectation**, then the sequence $\{\boldsymbol{w}_k\}$ can be guided toward a minimizer of R_n.

Batch Approach: batch gradient, steepest descent, full gradient method:

$$\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k - \alpha_k \nabla R_n(\boldsymbol{w}_k) = \boldsymbol{w}_k - \frac{\alpha_k}{n} \sum_{i=1}^{n} \nabla f_i(\boldsymbol{w}_k)$$

- more expensive
- can use all deterministic gradient-based optimization methods
- the sum structure opens up to parallelization

Analogues in simulation: stochastic approximation (SA) and sample average approximation (SAA)

Stochastic Gradient

- In case of redundancy using all the sample data in every iteration is inefficient
- Comparison of the performance of a batch L-BFGS method on number of evaluations of a sample gradient $\nabla f_{i_k}(\boldsymbol{w}_k)$.
- Each set of n consecutive accesses is called an **epoch**.
- The batch method performs only one step per epoch while SG performs n steps per epoch.

the fast initial improvement achieved by SG, followed by a drastic slowdown after 1 or 2 epochs, is common in practice

SG more sensitive to α_k and starting point

if more epochs, batch may become better

Rate of Convergence

Let $\{\boldsymbol{x}_k\}$ be a sequence in $\mathbb{R}^n$ that converges to $\boldsymbol{x}^*$.
The convergence is said to be **Q-linear** (quotient-linear) if there is a constant $r \in (0, 1)$ such that

$$\boxed{\frac{\|\boldsymbol{x}_{k+1} - \boldsymbol{x}^*\|}{\|\boldsymbol{x}_k - \boldsymbol{x}^*\|} \leq r \qquad \text{for all } k \text{ sufficiently large}}$$

ie, the distance to the solution $\boldsymbol{x}^*$ decreases at each iteration by at least a constant factor bounded away from 1 (ie, < 1).

Example:
sequence $\{1 + (0.5)^k\}$ converges Q-linearly to 1, with rate $r = 0.5$.

Rate of Convergence

The convergence is said to be **Q-superlinear** if

$$\boxed{\lim_{k\to\infty} \frac{\|\boldsymbol{x}_{k+1} - \boldsymbol{x}^*\|}{\|\boldsymbol{x}_k - \boldsymbol{x}^*\|} = 0}$$

Example: the sequence $\{1 + k^{-k}\}$ converges superlinearly to 1.

An even more rapid convergence rate: The convergence is said to be **Q-quadratic** if

$$\boxed{\frac{\|\boldsymbol{x}_{k+1} - \boldsymbol{x}^*\|}{\|\boldsymbol{x}_k - \boldsymbol{x}^*\|^2} \leq M \qquad \text{for all } k \text{ sufficiently large}}$$

where M is a positive constant, not necessarily less than 1. Example: the sequence $\{1 + (0.5)^{2^k}\}$. The values of r and M depend not only on the algorithm but also on the properties of the particular problem. Regardless of these values a quadratically convergent sequence will always eventually converge faster than a linearly convergent sequence.

Rate of Convergence

Superlinear convergence (quadratic, cubic, quartic, etc) is regarded as fast and desirable, while sublinear convergence is usually impractical.

- Steepest descent algorithms converge only at a Q-linear rate, and when the problem is ill-conditioned the convergence constant r in is close to 1.
- Quasi-Newton methods for unconstrained optimization typically converge Q-superlinearly
- Newton's method converges Q-quadratically under appropriate assumptions.

Rate of Convergence

A slightly weaker form of convergence:
overall rate of decrease in the error, rather than the decrease over each individual step of the algorithm.
We say that convergence is **R-linear** (root-linear) if there is a sequence of nonnegative scalars $\{v_k\}$ such that

$$\|\boldsymbol{x}_k - \boldsymbol{x}^*\| \leq \{v_k\} \text{ for all } k, \text{ and } \{v_k\} \text{ converges Q-linearly to zero.}$$

Theoretical Motivations

- a batch approach can minimize R_n at a fast rate; e.g., if R_n is strongly convex. A batch gradient method, then there exists a constant $\rho \in (0,1)$ such that, for all $k \in \mathbb{N}$, the training error follows **linear convergence**

$$R_n(\boldsymbol{w}_k) - R_n^* \leq \mathcal{O}(\rho^k),$$

- rate of convergence of a basic stochastic method is slower than for a batch gradient; e.g., if R_n is strictly convex and each i_k is drawn uniformly from $\{1, \ldots, n\}$, then for all $k \in \mathbb{N}$, SG satisfies the **sublinear convergence property**

$$\mathbb{E}[R_n(\boldsymbol{w}_k) - R_n^*] = \mathcal{O}(1/k).$$

  neither the per-iteration cost nor the right-hand side depends on the sample set size n

- in a stochastic optimization setting, SG yields for the expected risk the same convergence rate once substituted $\nabla f_{i_k}(\boldsymbol{w}_k)$ replaced by $\nabla f(\boldsymbol{w}_k; \xi_k)$ with each ξ_k drawn independently according to the distribution P

$$\mathbb{E}[R(\boldsymbol{w}_k) - R^*] = \mathcal{O}(1/k).$$

  If $n \gg k$ up to iteration k minimizing R_n same as minimizing R

Beyond SG: Noise Reduction and Second-Order Methods

- on horizontal axis methods that try to improve rate of convergence
- on vertical axis, methods that try to overcome non-linearity and ill-conditioning

Mini-batch Approach small subset of samples, call it $\mathcal{S}_k \subseteq \{1, \ldots, n\}$, chosen randomly in each iteration:

$$\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k - \frac{\alpha_k}{|\mathcal{S}_k|} \sum_{i \in \mathcal{S}} \nabla f_i(\boldsymbol{w}_k)$$

due to the reduced variance of the stochastic gradient estimates, the method is easier to tune in terms of choosing the stepsizes $\{\alpha_k\}$.
dynamic sample size and gradient aggregation methods, both of which aim to improve the rate of convergence from sublinear to linear

Outline

17. Analysis of SG

Theoretical Analysis — Preliminaries

convergence properties and worst-case iteration complexity bounds.

$$F(\boldsymbol{w}) = \begin{cases} R_n(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\boldsymbol{w}) & \text{Empirical Risk} \\ R(\boldsymbol{w}) = \mathbb{E}_{\xi}[f(\boldsymbol{w};\xi)] & \text{Expected Risk} \end{cases}$$

sampling uniformly with replacement from training set $\rightsquigarrow R_n$
sampling with $P(\xi)$ with replacement from training set $\rightsquigarrow R$.

Procedure SG(...);
Choose an initial iterate $\boldsymbol{w}_0$;
for $k = 0, 1, \ldots$ **do**
  Generate a realization of the random variable xi_k;
  Compute a stochastic vector $g(\boldsymbol{w}_k, \xi_k)$;
  Choose a stepsize $\alpha_k > 0$;
  Set the new iterate as $\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k - \alpha_k g(\boldsymbol{w}_k, \xi_k)$;

Theoretical Analysis — Preliminaries

ξ_k may represent a single sample or a mini-batch
g may represent a stochastic gradient (biased estimator of $\nabla F(\boldsymbol{w}_k)$ or a stochastic Netwon or quasi-Newton direction).

$$g(\boldsymbol{w}_k, \xi_k) = \begin{cases} \nabla f(\boldsymbol{w}_k; \xi_k) \\ \frac{1}{n_k} \sum_{i=1}^{n_k} \nabla f(\boldsymbol{w}_k; \xi_{k,i}) \\ H_k \frac{1}{n_k} \sum_{i=1}^{n_k} \nabla f(\boldsymbol{w}_k; \xi_{k,i}) \end{cases}$$

H_k a symmetric positive definite scaling matrix
α_k fixed stepsize or diminishing stepsizes
$\boldsymbol{w}_k$ can have influence on the sample selection (active learning)

Convergence Analysis – Assumptions

- Assumption 4.1 Lipschitz-continuous objective gradients
- Assumption 4.3 First and second moment limits. The objective function and SG (Algorithm 4.1) satisfy the following:
  - objective function to be bounded below by a scalar F_{inf} over the region explored by the algorithm.
  - in expectation, the vector $-g(\boldsymbol{w}_k, \xi_k)$ is a direction of sufficient descent for F from $\boldsymbol{w}_k$ with a norm comparable to the norm of the gradient
  - the variance of $g(\boldsymbol{w}_k, \xi_k)$ is restricted, but in a relatively minor manner.

$$\mathrm{Var}_{\xi_k}[g(\boldsymbol{w}_k, \xi_k)] \leq M + M_V \left\|\nabla F(\boldsymbol{w}_k)\right\|_2^2, \; M > 0, M_V > 0 \text{ for all } k \in \mathbb{N}$$

- Lemma: Markovian manner in the sense that $\boldsymbol{w}_{k+1}$ is a random variable that depends only on the iterate $\boldsymbol{w}_k$, the seed ξ_k, and the stepsize α_k and not on any past iterates.

Convergence Analysis – Assumptions

- Assumption 4.5 Strong convexity.
  The objective function $F : \mathbb{R}^d \to \mathbb{R}$ is **strongly convex** in that there exists a constant $c > 0$ such that

$$F(\bar{\boldsymbol{w}}) \geq F(\boldsymbol{w}) + \nabla F(\boldsymbol{w})^T(\bar{\boldsymbol{w}} - \boldsymbol{w}) + \frac{1}{2}c\,\|\bar{\boldsymbol{w}} - \boldsymbol{w}\|_2^2 \qquad \text{for all } (\bar{\boldsymbol{w}}, \boldsymbol{w}) \in \mathbb{R}^d \times \mathbb{R}^d$$

  or equivalently if there exists $c > 0$:

$$\nabla^2 F(\boldsymbol{w}) \succeq c$$

  (for univariate case: $f''(w) \geq c$), ie, grows at least quadratically.

  Hence, F has a unique minimizer, denoted as $\boldsymbol{w}^* \in \mathbb{R}^d$ with $F^* \stackrel{def}{=} F(w^*)$.

Convergence Analysis – Results

- Theorem 4.6 (Strongly Convex Objective, Fixed Stepsize).

- Theorem 4.7 (Strongly Convex Objective, Diminishing Stepsizes)
  SG with diminishing step size converges in expectation.
  - role of strong convexity
  - role of initial point
  - trade-offs of mini batches

- Theorem 4.8 (Nonconvex Objective, Fixed Stepsize)
  - While one cannot bound the expected optimality gap as in the convex case, inequality (4.28b) bounds the average norm of the gradient of the objective function observed on $\{\boldsymbol{w}_k\}$ visited during the first K iterations.
  - classical result for the full gradient method applied to nonconvex functions, namely, that the sum of squared gradients remains finite, implying that

$$\{\|\nabla F(\boldsymbol{w}_k)\|_2\} \to 0.$$

- Theorem 4.9 (Nonconvex Objective, Diminishing Stepsizes)
  - for the SG method with diminishing stepsizes, the expected gradient norms cannot stay bounded away from zero
  - the weighted average norm of the squared gradients converges to zero even if the gradients are noisy, (i.e., if $M > 0$ in the Variance upper bounding assumption) one can still conclude that the expected gradient norms cannot asymptotically stay far from zero.

Computational Complexity Analysis

- consider a big data scenario with an infinite supply of training examples, but a **limited computational time budget**. what type of algorithm — e.g., a simple SG or batch gradient method — would provide the best guarantees in terms of achieving a low expected risk?

- $\boldsymbol{w}^* \in \text{argmin}\, R(\boldsymbol{w})$; $\boldsymbol{w}_n \in \text{argmin}\, R_n(\boldsymbol{w})$, $\tilde{\boldsymbol{w}}_n$ approximate empirical risk minimizer returned by a given optimization algorithm at $\mathcal{T}_{max}$

- The tradeoffs associated with this scenario can be formalized as choosing the family of prediction functions $\mathcal{H}$, the number of examples n, and the optimization accuracy $\epsilon \stackrel{def}{=} E[R_n(\tilde{\boldsymbol{w}}_n) - R_n(\boldsymbol{w}_n)]$ in order to minimize the total error:

$$\underset{\mathcal{H}, n \in \mathbb{N}, \epsilon}{\text{minimize}}\; E[R(\tilde{\boldsymbol{w}}_n)] = \overbrace{R(\boldsymbol{w}^*)}^{\mathcal{E}_{app}(\mathcal{H})} + \overbrace{E[R(\boldsymbol{w}_n) - R(\boldsymbol{w}^*)]}^{\mathcal{E}_{est}(\mathcal{H}, n)} + \overbrace{E[R(\tilde{\boldsymbol{w}}_n) - R(\boldsymbol{w}_n)]}^{\mathcal{E}_{opt}(\mathcal{H}, n, \epsilon)}$$

$$\text{subject to } \mathcal{T}(n, \epsilon) \leq \mathcal{T}_{max}$$

Computational Complexity Analysis

- SG, with its sublinear rate of convergence, is more efficient for large-scale learning than (full, batch) gradient-based methods that have a linear rate of convergence.

- reducing the optimization error $\mathcal{E}_{opt}(H, n, \epsilon)$ (evaluated with respect to R rather than R_n) one might need to make up for the additional computing time by: (i) reducing the sample size n, potentially increasing the estimation error $\mathcal{E}_{est}(H, n)$; or (ii) simplifying the function family $\mathcal{H}$, potentially increasing the approximation error $\mathcal{E}_{app}(\mathcal{H})$.

Computational Complexity Analysis

Keep fixed $\mathcal{H}$ carrying out a worst-case analysis on the influence of the sample size n and optimization tolerance ϵ, which together only influence the estimation and optimization errors.

	Batch	Stochastic
$\mathcal{T}(n, \epsilon)$	$\sim n \log\left(\frac{1}{\epsilon}\right)$	$\frac{1}{\epsilon}$
$\mathcal{E}^*$	$\sim \frac{\log(\mathcal{T}_{\max})}{\mathcal{T}_{\max}} + \frac{1}{\mathcal{T}_{\max}}$	$\frac{1}{\mathcal{T}_{\max}}$

A stochastic optimization algorithm performs better that batch stochastic in terms of expected error

Large gap between asymptotical behavior and practical realities.

Remarks

- Fragility of the Asymptotic Performance of SG
  ok if objective function it includes a squared L_2-norm regularizer (related to constant c) but regularization parameter should be lowered when the number of samples increases.

- SG good for GPUs but ill-conditioning erodes efficiency of SG

- Distributed computing not working with basic SG because of too frequent updates of $\boldsymbol{w}$, more promising with mini-batch.

- Alternatives with Faster Convergence: minimizing empirical risk R_n there is information from previous gradients.
  - gradient aggregation methods
  - dynamic sampling approach

12. Beyond Stochastic Gradient

Noise Reduction Methods

- SG as the ideal optimization approach for large-scale applications.
- SG suffers from the adverse effect of **noisy gradient estimates**.
  - when fixed stepsizes are used it prevents SG from converging to the solution
  - when a diminishing stepsize sequence $\{\alpha_k\}$ is employed it leads to a slow, sublinear rate of convergence.
- Remedies:

Overview

Achieve linear rate of convergence to the optimal value using a fixed stepsize.

- **Dynamic sampling methods** achieve noise reduction by gradually increasing the mini-batch size used in the gradient computation, thus employing increasingly more accurate gradient estimates as the optimization process proceeds.

- **Gradient aggregation methods** improve the quality of the search directions by storing gradient estimates corresponding to samples employed in previous iterations, updating one (or some) of these estimates in each iteration, and defining the search direction as a weighted average of these estimates.

Rate of convergence remains sublinear but reduces variance of iterates

- **iterate averaging methods** maintain an average of iterates computed during the optimization process and employes a more aggressive stepsize sequence–of order $O(1/\sqrt{k})$ rather than $O(1/k)$.

Reducing Noise at a Geometric Rate

rate of decrease in noise that allows a stochastic-gradient-type method to converge at a linear rate.

Consequence of Lipschitz assumption with ℓ constant:

$$\mathbb{E}_{\xi_k}[F(\boldsymbol{w}_{k+1})] - F(\boldsymbol{w}_k) \leq -\alpha_k \nabla F(\boldsymbol{w}_k)^T \mathbb{E}_{\xi_k}[g(\boldsymbol{w}_k, \xi_k)] + \frac{1}{2}\alpha_k^2 \ell \mathbb{E}_{\xi_k}[\|g(\boldsymbol{w}_k, \xi_k)\|_2^2]$$

We want to make the left hand side small (sequence of expected optimality gaps).

<u>Theorem 5.1</u> (Strongly Convex Objective, Noise Reduction)
The SG method with a fixed stepsize $\bar{\alpha}$ and previous assumptions plus a variance of the stochastic vectors that decreases geometrically

$$\mathrm{Var}_{\xi_k}[g(\boldsymbol{w}_k, \xi_k)] \leq M\zeta^{k-1}$$

has a sequence of expected optimality gaps that vanishes at a linear rate:

$$\mathbb{E}[F(\boldsymbol{w}_k) - F^*] \leq \omega\rho^{k-1}$$

Dynamic Sample Size Methods

Can we design efficient optimization methods attaining the critical bound on the variance?

Mini-batch stochastic gradient:

$$\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k - \bar{\alpha} g(\boldsymbol{w}_k, \xi_k)$$

where the stochastic directions are computed for some $\tau > 1$ as

$$g(\boldsymbol{w}_k, \xi_k) \stackrel{def}{=} \frac{1}{n_k} \sum_{i \in \mathcal{S}_k} \nabla f(\boldsymbol{w}_k; \xi_{k,i}) \qquad \text{with } n_k \stackrel{def}{=} |\mathcal{S}_k| = \lceil \tau^{k-1} \rceil.$$

the mini-batch size increases geometrically as a function of the iteration counter k

Corollary 5.2. Let $\{\boldsymbol{w}_k\}$ be the iterates generated with unbiased gradient estimates, i.e., $\mathbb{E}_{\xi_{k,i}}[\nabla f(\boldsymbol{w}_k; \xi_{k,i})] = \nabla F(\boldsymbol{w}_k)$ for all $k \in \mathbb{N}$ and $i \in \mathcal{S}_k$. Then, the variance condition is satisfied, and if all other assumptions of Theorem 5.1 hold, then the expected optimality gap vanishes linearly.

Dynamic Sample Size Methods

Note: we described a method as linearly convergent but the per-iteration cost increases without bound.

Recall that SG method needs $\mathcal{T}(n, \epsilon) \leq 1/\epsilon$ evaluations to gurantee $\mathbb{E}[F(\boldsymbol{w}_k) - F^*] \leq \epsilon$

<u>Theorem 5.3</u> Suppose that the dynamic sampling SG method is run with a stepsize $\bar{\alpha}$ satisfying "some" bounds and some τ. In addition, suppose that all previous Assumptions hold. Then, the total number of evaluations of a stochastic gradient of the form $\nabla f(\boldsymbol{w}_k; \xi_{k,i})$ required to obtain $\mathbb{E}[F(\boldsymbol{w}_k) - F^*] \leq \epsilon$ is $O(\epsilon^{-1})$.

Dynamic Sample Size Guidelines

Given the rate of convergence of a batch optimization algorithm on strongly convex functions (i.e., linear, superlinear, etc.), what should be the sampling rate so that the overall algorithm is **efficient** in the sense that it results in the lowest computational complexity?

- if the optimization method has a sublinear rate of convergence, then there is no sampling rate that makes the algorithm "efficient";
- if the optimization algorithm is linearly convergent, then the sampling rate must be geometric (with restrictions on the constant in the rate) for the algorithm to be "efficient";
- for superlinearly convergent methods, increasing the sample size at a rate that is slightly faster than geometric will yield an "efficient" method.

Design in Practice

- presetting the sampling rate, ie, $\tau > 1$ before running the optimization algorithm, requires some experimentation. Care must be put in preventing the full sample set from being employed too soon

- adaptive mechanisms to produce descent directions sufficiently often
  - any direction $g(\boldsymbol{w}_k, \xi_k)$ is a descent direction for F at $\boldsymbol{w}_k$ if, for some $\chi \in [0, 1)$, one has

    $$\delta(\boldsymbol{w}_k, \xi_k) \stackrel{def}{=} \|g(\boldsymbol{w}_k, \xi_k) - \nabla F(wk)\|_2 \leq \chi \|g(\boldsymbol{w}_k, \xi_k)\|_2$$

    verifying the inequality may be costly because involves the evaluation of $\nabla F(\boldsymbol{w}_k)$, one can estimate the left-hand side $\delta(\boldsymbol{w}_k, \xi_k)$, and then choose n_k so it holds sufficiently often.
  - The sample variance obtained by sampling without replacements is bounded above by $\chi^2 \|g(\boldsymbol{w}_k, \xi_\epsilon)\|_2^2$
  - If this condition is not satisfied, then increase the sample size to a size that one might predict would satisfy such a condition.
  - no guarantee that the size n_k increases at a geometric rate. Remedy: if the adaptive increasesthe sampling rate more slowly than a preset geometric sequence, then a growth in the sample size is imposed.

Gradient Aggregation

- Rather than compute increasingly more **new** stochastic gradient information in each iteration, achieve a lower variance by **reusing and/or revising** previously computed information
- achieve a linear rate of convergence on strongly convex problems.
- improved rate is achieved primarily by either an increase in computation or an increase in storage.
- works on finite sums like R_n

SVRG

Procedure SVRG ; # Methods for Minimizing an Empirical Risk R_n

Choose an initial iterate $\boldsymbol{w}_1 \in \mathbb{R}^d$, stepsize $\alpha > 0$ and a positive integer m;

for $k = 1, 2, \ldots$ **do**

  Compute the batch gradient $\nabla R_n(\boldsymbol{w}_k)$;

  Initialize $\tilde{\boldsymbol{w}}_1 \leftarrow \boldsymbol{w}_k$;

  for $j = 1, \ldots, m$ **do**

    $\tilde{\boldsymbol{g}}_j \leftarrow \nabla f_{i_j}(\tilde{\boldsymbol{w}}_j) - (\nabla f_{i_j}(\boldsymbol{w}_k) - \nabla R_n(\boldsymbol{w}_k))$; # $\nabla R_n(\boldsymbol{w}_k)$ from batch gradient

    $\tilde{\boldsymbol{w}}_{j+1} \leftarrow \tilde{\boldsymbol{w}}_j - \alpha \tilde{\boldsymbol{g}}_j$;

  Option (a): Set $\boldsymbol{w}_{k+1} = \tilde{\boldsymbol{w}}_{m+1}$;

  Option (b): Set $\boldsymbol{w}_{k+1} = \frac{1}{m} \sum_{j=1}^{m} \tilde{w}_{j+1}$;

  Option (c): Choose j uniformly from $\{1, \ldots, m\}$ and set $\boldsymbol{w}_{k+1} = \tilde{\boldsymbol{w}}_{j+1}$;

- since $E_{i_j}[\nabla f_{i_j}(\boldsymbol{w}_k)] = \nabla R_n(\boldsymbol{w}_k)$, one can view $\nabla f_{i_j}(\boldsymbol{w}_k) - \nabla R_n(\boldsymbol{w}_k)$ as the bias in the gradient estimate $\nabla f_{i_j}(\boldsymbol{w}_k)$.
- sampled gradient $\nabla f_{i_j}(\tilde{\boldsymbol{w}}_j)$ is corrected based on a perceived bias. Overall, $\tilde{\boldsymbol{g}}_j$ represents an unbiased estimator of $\nabla R_n(\tilde{\boldsymbol{w}}_j)$, but with a variance that one can expect to be smaller than as in simple SG

SAGA

in each iteration, it computes a stochastic vector $\boldsymbol{g}_k$ as the average of stochastic gradients evaluated at previous iterates.

Procedure SAGA ; # Method for Minimizing an Empirical Risk R_n

Choose an initial iterate $\boldsymbol{w}_1 \in \mathbb{R}^d$ and stepsize $\alpha > 0$;

for $i = 1, \ldots, n$ **do**

  Compute $\nabla f_i(\boldsymbol{w}_1)$;

  Store $\nabla f_i(\boldsymbol{w}_{[i]}) \leftarrow \nabla f_i(\boldsymbol{w}_1)$; # $\boldsymbol{w}_{[i]}$ represents the latest iterate at which ∇f_i

for $k = 1, 2, \ldots$ **do**

  Choose j uniformly in $\{1, \ldots, n\}$;

  Compute $\nabla f_j(\boldsymbol{w}_k)$;

  Set $\boldsymbol{g}_k \leftarrow \nabla f_j(\boldsymbol{w}_k) - \nabla f_j(\boldsymbol{w}_{[j]}) + \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\boldsymbol{w}_{[i]})$;

  Store $\nabla f_j(\boldsymbol{w}_{[j]}) \leftarrow \nabla f_j(\boldsymbol{w}_k)$;

  Set $\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k - \alpha \boldsymbol{g}_k$;

As in SVRG, the method employs unbiased gradient estimates, but with variances that are expected to be less than the stochastic gradients that would be employed in a basic SG routine

SAGA

- Same per-iteration costs as basic SG
- on strongly convex R_n can achieve a linear rate of convergence but needs knowledge of at least ℓ.
- More effective initialization instead of evaluating all the gradients $\{\nabla f_i\}_{i=1}^n$ at the initial point. For example, one could perform one epoch of simple SG, or one can assimilate iterates one-by-one and compute $\boldsymbol{g}_k$ only using the gradients available up to that point.
- SAGA needs to store n stochastic gradient vectors
- for very large n, gradient aggregation methods are comparable to batch algorithms and therefore cannot beat SG in this regime

Iterated Averaging Methods

- for minimizing a continuously differentiable F with unbiased gradient estimates, the idea is to employ the iteration:

$$\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k - \alpha g(\boldsymbol{w}_k, \xi_k)$$

$$\tilde{\boldsymbol{w}}_{k+1} \leftarrow \frac{1}{k+1} \sum_{j=1}^{k+1} \boldsymbol{w}_j$$

  where $\{\tilde{\boldsymbol{w}}_k\}$ has no effect on the computation of the SG iterate sequence $\{\boldsymbol{w}_k\}$

- with stepsizes diminishing at a slow rate of $\mathcal{O}(1/(k^a))$ for some $a \in (\frac{1}{2}, 1)$ on strongly convex objectives, yields that $\mathbb{E}[\|\boldsymbol{w}_k - \boldsymbol{w}^*\|_2^2] = \mathcal{O}(1/(k^a))$ while $\mathbb{E}[\|\tilde{\boldsymbol{w}}_k - \boldsymbol{w}^*\|_2^2] = \mathcal{O}(1/k)$.

- in certain cases this combination of long steps and averaging yields an optimal constant in $\mathbb{E}[\|\tilde{\boldsymbol{w}}_k - \boldsymbol{w}^*\|_2^2]$ in the sense that no rescaling of the steps—through multiplication with a positive definite matrix (second order methods) can improve the asymptotic rate or constant.

Second Order Methods

- Address the adverse effects of high nonlinearity and ill-conditioning of the objective function through the use of second-order information.

- improve convergence rates of batch methods or the constants involved in the sublinear convergence rate of stochastic methods

- First-order methods are **not scale invariant**. Consider:
  F continously differentiable function $F : \mathbb{R}^d \to \mathbb{R}$

  $$\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k - \alpha_k \nabla F(\boldsymbol{w}_k)$$

  linear transformation of the variables $\{\boldsymbol{w}_k\} = \{B\tilde{\boldsymbol{w}}_k\}$. $\min_{\tilde{\boldsymbol{w}}} F(B\tilde{\boldsymbol{w}}_k)$

  $$\tilde{\boldsymbol{w}}_{k+1} \leftarrow \tilde{\boldsymbol{w}}_k - \alpha_k B \nabla F(B\tilde{\boldsymbol{w}}_k) \qquad \Longrightarrow \qquad \boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k - \alpha_k B^2 \nabla F(\boldsymbol{w}_k)$$

  They will perform differently. With $\alpha = 1$ and $B = (\nabla^2 F(\boldsymbol{w}_1))^{-1/2}$ we get Newton's method

- Newton's method achieves a quadratic rate of convergence if $\boldsymbol{w}_1$ is sufficiently close to a strong minimizer. On the other hand, stochastic methods like the SG method cannot achieve a convergence rate that is faster than sublinear, regardless of the choice of B.

- careful use of successive re-scalings based on (approximate) second-order derivatives can be beneficial between the stochastic and batch regimes.

Hessian-Free Inexact Newton Methods

- Newton's method:

  $$\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k + \alpha_k \boldsymbol{s}_k$$

  where $\boldsymbol{s}_k$ satisfies $\nabla^2 F(\boldsymbol{w}_k)\boldsymbol{s}_k = -\nabla F(\boldsymbol{w}_k)$.

- one can solve the linear system inexactly through an iterative approach such as the conjugate gradient (CG) method.

- By ensuring that the linear solves are accurate enough, such an inexact Newton-CG method can enjoy a superlinear rate of convergence

- For a smooth objective function F, one can compute $\nabla^2 F(\boldsymbol{w})\boldsymbol{d}$ at a cost that is a small multiple of the cost of evaluating ∇F, and without forming the Hessian, which would require $\mathcal{O}(d^2)$ storage

- exploit structure of risk measures

- iterations are more tolerant to noise in the hessian estimate than it is to noise in the gradient estimate

- employs a smaller, conditionally (given $\boldsymbol{w}_k$) uncorrelated, sample for defining the Hessian than for the stochastic gradient estimate

- can be combined with a backtracking (Armijo) line search or trust region

- (subsampled) Hessian-vector products can be computed efficiently in ML tasks

Example 6.2. *Consider a binary classification problem where the training function is given by the logistic loss with an ℓ_2-norm regularization parameterized by $\lambda > 0$:*

$$R_n(w) = \frac{1}{n}\sum_{i=1}^{n} \log(1 + \exp(-y_i w^T x_i)) + \frac{\lambda}{2}\|w\|^2. \tag{6.8}$$

A (subsampled) Hessian-vector product can be computed efficiently by observing that

$$\nabla^2 f_{\mathcal{S}_k^H}(w_k; \xi_k^H) d = \frac{1}{|\mathcal{S}_k^H|} \sum_{i \in \mathcal{S}_k^H} \frac{\exp(-y_i w^T x_i)}{(1 + \exp(-y_i w^T x_i))^2} (x_i^T d) x_i + \lambda d.$$

- Stochastic Quasi-Newton Methods:
  Like BFGS

- Gauss-Newton Methods
  constructs an approximation to the Hessian using only first-order information, and this approximation is guaranteed to be positive semidefinite, even when the full Hessian itself may be indefinite.
  The price to pay for this convenient representation is that it ignores second-order interactions between elements of the parameter vector $\boldsymbol{w}$, which might mean a loss of curvature information that could be useful for the optimization process.

Summary

- Ways to cope with the problems in machine learning
- SG might not be the best choice for parallelization
- How about other methods like CMA-ES?

13. Constrained Optimization

Constrained Optimization

- Minimizing an objective subject to design point restrictions called **constraints**
- A variety of techniques transform constrained optimization problems into unconstrained problems
- New optimization problem statement

$$\begin{aligned} \underset{\boldsymbol{x}}{\text{minimize}} \quad & f(\boldsymbol{x}) \\ \text{subject to} \quad & \boldsymbol{x} \in \mathcal{X} \end{aligned}$$

- The set $\mathcal{X} \subset \mathbb{R}$ is called the **feasible set**

Constrained Optimization

Constraints that bound feasible set can change the optimizer

$$\underset{x}{\text{minimize}}\ f(x)$$

$$\text{subject to } x \in [a, b]$$

Constraint Types

- Generally, constraints are formulated using two types:
  1. **Equality constraints**: $h(\boldsymbol{x}) = 0$
  2. **Inequality constraints**: $g(\boldsymbol{x}) \leq 0$
- Any optimization problem can be written as

$$\begin{aligned} \underset{\boldsymbol{x}}{\text{minimize}} \quad & f(\boldsymbol{x}) \\ \text{subject to} \quad & g_i(\boldsymbol{x}) \leq 0 \text{ for all } i \text{ in } \{1, \ldots, m\} \\ & h_j(\boldsymbol{x}) = 0 \text{ for all } j \text{ in } \{1, \ldots, \ell\} \end{aligned}$$

$$\begin{aligned} \underset{\boldsymbol{x}}{\text{minimize}} \quad & f(\boldsymbol{x}) \\ \text{subject to} \quad & \boldsymbol{g}(\boldsymbol{x}) \leq 0 \\ & \boldsymbol{h}(\boldsymbol{x}) = 0 \end{aligned}$$

f and the functions h and g are all smooth, real-valued functions on a subset of Re^n

Transformations to Remove Constraints

- If necessary, some problems can be reformulated to incorporate constraints into the objective function

- If x is constrained between a and b

$$x = t_{a,b}(\hat{x}) = \frac{b+a}{2} + \frac{b-a}{2}\left(\frac{2\hat{x}}{1+\hat{x}^2}\right)$$

Transformations to Remove Constraints

Example

$$\underset{x}{\text{minimize}} \;\; x\sin(x)$$
$$\text{subject to} \;\; 2 \leq x \leq 6$$

$$\underset{\hat{x}}{\text{minimize}} \; t_{2,6}(\hat{x})\sin(t_{2,6}(\hat{x}))$$

$$\underset{\hat{x}}{\text{minimize}} \; \left(4 + 2\left(\frac{2\hat{x}}{1+\hat{x}^2}\right)\right)\sin\left(4 + 2\left(\frac{2\hat{x}}{1+\hat{x}^2}\right)\right)$$

Transformations to Remove Constraints

Example

$$\begin{aligned} &\underset{x}{\text{minimize}} \;\; f(\boldsymbol{x}) \\ &\text{subject to} \;\; h(\boldsymbol{x}) = x_1^2 + x_2^2 + \ldots + x_n^2 - 1 = 0 \end{aligned}$$

- Solve for one of the variables to eliminate constraint:

$$x_n = \pm\sqrt{1 - x_1^2 - x_2^2 - \ldots - x_{n-1}^2}$$

- Transformed, unconstrained optimization problem:

$$\underset{x}{\text{minimize}} \;\; \left(\left[x_1, x_2, \ldots, x_{n-1}, \pm\sqrt{1 - x_1^2 - x_2^2 - \ldots - x_{n-1}^2}\right]\right)$$

Lagrangian Relaxation

- With only equality constraints, critical points (local minima, global minima, or saddle points optimal) where gradient of f and the gradient of h are aligned
- The method of Lagrangian relaxation is used to optimize a function subject to (equality) constraints
- Lagrangian multipliers refer to the variables introduced by the method denoted by λ

$$\begin{aligned} \underset{\boldsymbol{x}}{\text{minimize}} \quad & f(\boldsymbol{x}) \\ \text{subject to} \quad & h(\boldsymbol{x}) = 0 \end{aligned}$$

1. Form **Lagrangian relaxation**

$$\mathcal{L}(\boldsymbol{x}, \lambda) = f(\boldsymbol{x}) - \lambda h(\boldsymbol{x})$$

2. Set $\nabla_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}, \lambda) = 0$ and $\nabla_{\lambda} \mathcal{L}(\boldsymbol{x}, \lambda) = 0$ to get

$$\nabla f(\boldsymbol{x}) = \lambda \nabla h(\boldsymbol{x}) \qquad h(\boldsymbol{x}) = 0$$

3. solve for $\boldsymbol{x}$ and λ

Example

$$\text{minimize} \quad -\exp\left(-\left(x_1 x_2 - \frac{3}{2}\right)^2 - \left(x_2 - \frac{3}{2}\right)^2\right)$$

$$\text{subject to } x_1 - x_2^2 = 0$$

Lagrangian Relaxation

Intuitively, the method of Lagrange multipliers finds the point x^* where the constraint function is orthogonal to the gradient

Lagrangian Relaxation with Inequality Constraints

$$\begin{aligned} \underset{\boldsymbol{x}}{\text{minimize}} \;\; & f(\boldsymbol{x}) \\ \text{subject to} \;\; & g(\boldsymbol{x}) \leq 0 \end{aligned}$$

- If solution lies at the constraint boundary, the constraint is called **active**, and the Lagrangian condition holds for a non-negative constant μ:

$$\nabla f(\boldsymbol{x}) + \mu \nabla g(\boldsymbol{x}) = 0$$

- If the solution lies within the boundary, the constraint is called **inactive**, and the optimal solution simply lies where

$$\nabla f(\boldsymbol{x}) = 0$$

that is, the Lagrangian condition holds with $\mu = 0$

Lagrangian Relaxation with Inequality Constraints

$$\underset{\boldsymbol{x}}{\text{minimize}} \;\; f(\boldsymbol{x})$$
$$\text{subject to} \;\; g(\boldsymbol{x}) \leq 0$$

- We create the Lagrangian relaxation such that it goes to ∞ outside the feasibility set ($g(\boldsymbol{x}) \not\leq 0$)):

$$\mathcal{L}_\infty(\boldsymbol{x}) = f(\boldsymbol{x}) + \infty(g(\boldsymbol{x}) > 0)$$

impractical: discontinuous and nondifferentiable.

- Instead, for $\mu > 0$:

$$\mathcal{L}(\boldsymbol{x}, \mu \geq 0) = f(\boldsymbol{x}) + \mu g(\boldsymbol{x})$$

$$\mathcal{L}_\infty(\boldsymbol{x}) = \underset{\mu \geq 0}{\text{maximize}} \; \mathcal{L}(\boldsymbol{x}, \mu)$$

for $\boldsymbol{x}$ infeasible, $\mathcal{L}_\infty(\boldsymbol{x}) = \infty$; for $\boldsymbol{x}$ feasible, $\mathcal{L}_\infty(\boldsymbol{x}) = f(\boldsymbol{x})$

- The new optimization problem becomes

$$\underset{\boldsymbol{x}}{\text{minimize}} \; \underset{\mu \geq 0}{\text{maximize}} \; \mathcal{L}(\boldsymbol{x}, \mu)$$

This is called the **primal problem**

Necessary Conditions – KKT Conditions

$$\begin{aligned} \underset{\boldsymbol{x}}{\text{minimize}} \quad & f(\boldsymbol{x}) \\ \text{subject to} \quad & \boldsymbol{g}(\boldsymbol{x}) \leq 0 \\ & \boldsymbol{h}(\boldsymbol{x}) = 0 \end{aligned}$$

Any critical point $\boldsymbol{x}^*$ must satisfy the **Karush-Kuhn-Tucker conditions**

1. primal feasibility: $\boldsymbol{g}(\boldsymbol{x}^*) \leq 0$ and $\boldsymbol{h}(\boldsymbol{x}^*) = 0$
2. dual feasibility: penaliztion is towards feasibility $\boldsymbol{\mu} \geq 0$
3. complementary slackness: either μ_i or $g_i(\boldsymbol{x}^*)$ is zero.

$$\mu_i g_i(\boldsymbol{x}^*) = 0, \text{ for } i = 1, \ldots, m.$$

4. stationarity: objective function tanget to each active constraint

$$\nabla f(\boldsymbol{x}^*) + \sum_i \mu_i \nabla g_i(\boldsymbol{x}^*) + \sum_j \lambda_j \nabla h_j(\boldsymbol{x}^*) = 0$$

Necessary Conditions – KKT Conditions

Particular cases

- f concave, $\boldsymbol{g}$ convex: then KKT are also sufficient
- Patological cases

In vector form:

$$\begin{cases} \nabla f(\boldsymbol{x}^*) + \boldsymbol{\mu} \cdot \nabla \boldsymbol{g}(\boldsymbol{x}^*) + \boldsymbol{\lambda} \cdot \nabla \boldsymbol{h}(\boldsymbol{x}^*) = 0 \\ \boldsymbol{\mu} \cdot \boldsymbol{g}(\boldsymbol{x}^*) = 0 \\ \boldsymbol{g}(\boldsymbol{x}^*) \leq 0, \ \boldsymbol{h}(\boldsymbol{x}^*) = 0 \\ \boldsymbol{\mu} \geq 0 \end{cases}$$

Duality

- Generalized Lagrangian Relaxation:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) + \sum_i \mu_i g_i(\boldsymbol{x}) + \sum_j \lambda_j h_j(\boldsymbol{x})$$

- the primal form is

$$\underset{\boldsymbol{x}}{\text{minimize}} \; \underset{\boldsymbol{\mu} \geq 0, \boldsymbol{\lambda}}{\text{maximize}} \; \mathcal{L}(\boldsymbol{x}, \boldsymbol{\mu}.\boldsymbol{\lambda})$$

- Reversing the order of operations leads to the **dual form**

$$\underset{\boldsymbol{\mu} \geq 0, \boldsymbol{\lambda}}{\text{maximize}} \; \underset{\boldsymbol{x}}{\text{minimize}} \; \mathcal{L}(\boldsymbol{x}, \boldsymbol{\mu}, \boldsymbol{\lambda})$$

- In some cases, the dual problem is easier to solve computationally than the original problem. In other cases, the dual can be used to obtain easily a lower bound on the optimal value of the objective for the primal problem. The dual has also been used to design algorithms for solving the primal problem.

Duality

Theorem (Max-min inequality)

For any function $f : Z \times W \to \mathbb{R}$,

$$\sup_{z \in Z} \inf_{w \in W} f(z, w) \leq \inf_{w \in W} \sup_{z \in Z} f(z, w) \ .$$

Proof: see wikipedia □

- When f, W, and Z are convex the inequality becomes equality and we have a strong max–min property (or a saddle-point property).
- For us:

$$\underset{\boldsymbol{\mu} \geq 0, \boldsymbol{\lambda}}{\text{maximize}} \ \underset{\boldsymbol{x}}{\text{minimize}} \ \mathcal{L}(\boldsymbol{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) \leq \underset{\boldsymbol{x}}{\text{minimize}} \ \underset{\boldsymbol{\mu} \geq 0, \boldsymbol{\lambda}}{\text{maximize}} \ \mathcal{L}(\boldsymbol{x}, \boldsymbol{\mu}.\boldsymbol{\lambda})$$

- Therefore, the solution to the dual problem d^* is a lower bound to the primal solution p^*
- The inner part of the dual problem can be used to define the **dual function** or **dual objective**

$$\mathcal{D}(\boldsymbol{\mu} \geq 0, \boldsymbol{\lambda}) = \underset{\boldsymbol{x}}{\text{minimize}} \ \mathcal{L}(\boldsymbol{x}, \boldsymbol{\mu}, \boldsymbol{\lambda})$$

Duality

- The dual function is concave. Gradient ascent on a concave function always converges to the global maximum.
- **Dual Problem**: $\max \mathcal{D}(\lambda)$ subject to $\lambda \geq 0$
- Optimizing the dual problem is easy whenever minimizing the Lagrangian with respect to x is easy.
- For any $\boldsymbol{\mu} \geq 0$ and any $\boldsymbol{\lambda}$, we have

$$\mathcal{D}(\boldsymbol{\mu} \geq 0, \boldsymbol{\lambda}) \leq p^*$$

- The difference between dual and primal solutions d^* and p^* is called the **duality gap**
- Showing zero-duality gap is a "certificate" of optimality

Penalty methods

- Penalty methods are a way of reformulating a constrained optimization problem as an unconstrained problem by penalizing the objective function value when constraints are violated

Example

$$\begin{aligned} \underset{\boldsymbol{x}}{\text{minimize}} \quad & f(\boldsymbol{x}) \\ \text{subject to} \quad & \boldsymbol{g}(\boldsymbol{x}) \leq 0 \\ & \boldsymbol{h}(\boldsymbol{x}) = 0 \end{aligned}$$

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}) + \rho \cdot p_{count}(\boldsymbol{x}) \\ \text{s.t.} \quad & p_{count}(\boldsymbol{x}) = \sum_i (g_i(\boldsymbol{x}) > 0) + \sum_j (h_j(\boldsymbol{x}) \neq 0) \end{aligned}$$

Penalty Methods

Procedure penalty_method;
Input: $f, p, \boldsymbol{x}, k_{max}; \rho = 1, \gamma = 2$
Output: $\boldsymbol{x}$ solution
for k in $1, \ldots, k_{max}$ **do**
  $\boldsymbol{x} \leftarrow \text{minimize}_{\boldsymbol{x}}\{f(\boldsymbol{x}) + \rho \cdot p(\boldsymbol{x})\}$;
  $\rho \leftarrow \rho \cdot \gamma$;
  if $p(\boldsymbol{x}) = 0$ **then**
    return $\boldsymbol{x}$;
return $\boldsymbol{x}$;

Penalty methods

- Count penalty:

$$p_{count}(\boldsymbol{x}) = \sum_i (g_i(\boldsymbol{x}) > 0) + \sum_j (h_j(\boldsymbol{x}) \neq 0)$$

- Quadratic penalty:

$$p_{quadratic}(\boldsymbol{x}) = \sum_i \max(g_i(\boldsymbol{x}), 0)^2 + \sum_j h_j(\boldsymbol{x})^2$$

- Mixed Penalty:

$$p_{mixed}(\boldsymbol{x}) = \rho_1 p_{count}(\boldsymbol{x}) + \rho_2 p_{quadratic}(\boldsymbol{x})$$

Augmented Lagrange Method

- Adaptation of penalty method for equality constraints

$$p_{Lagrangian}(\boldsymbol{x}) \stackrel{def}{=} \frac{1}{2}\rho \sum_i (h_i(\boldsymbol{x}))^2 - \sum_i \lambda_i h_i(\boldsymbol{x})$$

Procedure augmented_lagrange_method;
Input: $f, h, \boldsymbol{x}, k_{max}; \rho = 1, \gamma = 2)$
$\boldsymbol{\lambda} \leftarrow 0$;
for k in $1, \ldots, k_{max}$ **do**
  $p \leftarrow \left(x \mapsto \rho/2 \cdot \sum_i (h_i(\boldsymbol{x})^2) - \boldsymbol{\lambda} \cdot h(\boldsymbol{x})\right)$;
  $\boldsymbol{x} \leftarrow \text{minimize}_{\boldsymbol{x}}\{f(\boldsymbol{x}) + p(\boldsymbol{x})\}$;
  $\boldsymbol{\lambda} \leftarrow \boldsymbol{\lambda} - \rho \cdot \boldsymbol{h}(\boldsymbol{x})$;
  $\rho \leftarrow \rho \cdot \gamma$;
return $\boldsymbol{x}$;

- $\boldsymbol{\lambda}$ converges towards the Lagrangian multiplier

Interior Point Methods

- Also called **barrier methods**, interior point methods ensure that each step is feasible
- This allows premature termination to return a nearly optimal, feasible point
- Barrier functions are implemented similar to penalties but must meet the following conditions:
  1. Continuous
  2. Non-negative
  3. Approach infinity as x approaches boundary

Interior Point Methods

- Inverse Barrier:

$$p_{barrier}(\boldsymbol{x}) = -\sum_i \frac{1}{g_i(\boldsymbol{x})}$$

- Log Barrier:

$$p_{barrier}(\boldsymbol{x}) = -\sum_i \begin{cases} \log(-g_i(\boldsymbol{x})) & \text{if } g_i(\boldsymbol{x}) \geq -1 \\ 0 & \text{otherwise} \end{cases}$$

New optimization problem:

$$\underset{x}{\text{minimize}}\ f(\boldsymbol{x}) + \frac{1}{\rho} p_{barrier}(\boldsymbol{x})$$

Interior Point Methods

Procedure interior_point_method;
Input: $f, p, \boldsymbol{x}; \rho = 1, \gamma = 2, \epsilon = 0.001$
$\Delta \leftarrow \infty$;
while $\Delta > \epsilon$ **do**
  $\boldsymbol{x}' \leftarrow \text{minimize}_{\boldsymbol{x}}\{f(\boldsymbol{x}) + p(\boldsymbol{x})/\rho\}$;
  $\Delta \leftarrow \|\boldsymbol{x}' - \boldsymbol{x}\|$;
  $\boldsymbol{x} \leftarrow \boldsymbol{x}'$;
  $\rho \leftarrow \rho \cdot \gamma$;
return $\boldsymbol{x}$;

- Line searches $f(\boldsymbol{x} + \alpha \boldsymbol{d})$ are constrained to the interval $\alpha = [0, \alpha_u]$, where α_u is the step to the nearest boundary.
  In practice, α_u is chosen such that $\boldsymbol{x} + \alpha \boldsymbol{d}$ is just inside the boundary to avoid the boundary singularity.
- Needs an initial **feasible** solutions. Typically, found by solving:

$$\underset{\boldsymbol{x}}{\text{minimize}}\ p_{quadratic}(\boldsymbol{x})$$

Summary

- Constraints are requirements on the design points that a solution must satisfy
- Some constraints can be transformed or substituted into the problem to result in an unconstrained optimization problem
- Analytical methods using Lagrange multipliers yield the generalized Lagrangian and the necessary conditions for optimality under constraints
- A constrained optimization problem has a dual problem formulation that is easier to solve and whose solution is a lower bound of the solution to the original problem
- Penalty methods penalize infeasible solutions and often provide gradient information to the optimizer to guide infeasible points toward feasibility
- Interior point methods maintain feasibility but use barrier functions to avoid leaving the feasible set

14. Linear Constrained Optimization

Problem Formulation

- If an optimization problem has a linear objective and constraints, it is called a **linear programming problem (linear program, LP)**

- The general form of a linear program is:

$$
\begin{aligned}
\underset{\boldsymbol{x}}{\text{minimize}} \quad & \boldsymbol{c}^T \boldsymbol{x} \\
\text{subject to} \quad & A\boldsymbol{x} \leq \boldsymbol{b} \\
& D\boldsymbol{x} \geq \boldsymbol{e} \\
& F\boldsymbol{x} = \boldsymbol{g} \\
& \\
& \boldsymbol{x}, \boldsymbol{c} \in \mathbb{R}^n, \\
& A \in \mathbb{R}^{m \times n}, \boldsymbol{b} \in \mathbb{R}^m \\
& D \in \mathbb{R}^{p \times n}, \boldsymbol{e} \in \mathbb{R}^p \\
& F \in \mathbb{R}^{q \times n}, \boldsymbol{g} \in \mathbb{R}^q
\end{aligned}
$$

Numerical Example

$$
\begin{aligned}
\underset{x_1, x_2, x_3}{\text{minimize}} \quad & 2x_1 - 3x_2 + 7x_3 \\
\text{subject to} \quad & 2x_1 + 3x_2 - 8x_3 \leq 5 \\
& 4x_1 + x_2 + 3x_3 \leq 9 \\
& x_1 - 5x_2 - 3x_3 \geq -4 \\
& x_1 + x_2 + 2x_3 = 1
\end{aligned}
$$

Modelling in Linear Programming

Example

Given a set of items I, each item with a price p_i and a value v_i, i in I, select the subset of items that maximizes the total value collected subject to a total expense that does not exceed a given budget B.

$$\begin{aligned} \max \quad & \sum_{i \in I} p_i x_i \\ \text{s.t.} \quad & \sum_{i \in I} v_i x_i \leq B \\ & x_i \in \{0, 1\}, \quad \text{for all } i \text{ in } I \end{aligned}$$

Modelling in Linear Programming

Many problems can be converted into linear programs that have the same solution.

Example

$$\text{minimize } L_1 = \|A\boldsymbol{x} - \boldsymbol{b}\|_1$$

$$\begin{aligned} \min\ & 1^T \boldsymbol{s} \\ \text{s.t. } & A\boldsymbol{x} - \boldsymbol{b} \leq \boldsymbol{s} \\ & -(A\boldsymbol{x} - \boldsymbol{b}) \leq \boldsymbol{s} \end{aligned}$$

Example

$$\text{minimize } L_\infty = \|A\boldsymbol{x} - \boldsymbol{b}\|_\infty$$

$$\begin{aligned} \min\ & t \\ \text{s.t. } & A\boldsymbol{x} - \boldsymbol{b} \leq t1 \\ & -(A\boldsymbol{x} - \boldsymbol{b}) \leq t1 \end{aligned}$$

Problem Formulation

Every general form linear program can be rewritten more compactly in **standard form**

$$
\begin{aligned}
\underset{\boldsymbol{x}}{\text{minimize}} \quad & \boldsymbol{c}^T \boldsymbol{x} \\
\text{subject to} \quad & A\boldsymbol{x} \leq \boldsymbol{b} \\
& \boldsymbol{x} \geq 0 \\
& \\
& \boldsymbol{x}, \boldsymbol{c} \in \mathbb{R}^n, \\
& A \in \mathbb{R}^{m \times n}, \boldsymbol{b} \in \mathbb{R}^m
\end{aligned}
$$

Example

$$
\begin{aligned}
\text{minimize } & 5x_1 + 4x_2 \\
\text{s.t. } & 2x_1 + 3x_2 \leq 5 \\
& 4x_1 + x_2 \leq 11
\end{aligned}
$$

Problem Formulation

- Each inequality constraint defines a planar boundary of the feasible set called a **half-space**
- The set of inequality constraints define the intersection of multiple half-spaces forming a **convex set**
- Convexity of the feasible set, along with convexity of the objective function, implies that if we find a local feasible minimum, it is also a global feasible minimum.

$$\begin{aligned} \underset{\boldsymbol{x}}{\text{minimize}} \quad & \boldsymbol{c}^T \boldsymbol{x} \\ \text{subject to} \quad & A\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \geq 0 \end{aligned}$$

Half-Spaces and Supporting Hyperplanes

Problem Formulation

- How many solutions are there?

Problem Formulation

Linear programs are often solved in **equality form**

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \boldsymbol{c}^T \boldsymbol{x} \\ \text{subject to} \quad & A\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geq 0 \end{aligned}$$

$\boldsymbol{x}, \boldsymbol{c} \in \mathbb{R}^{2n+m}$,

$A \in \mathbb{R}^{m \times 2n+m}$,

$\boldsymbol{b} \in \mathbb{R}^m$

Simplex Algorithm

- Guaranteed to solve any feasible and bounded linear program
- Works on the equality form
- Assumes that rows of A are linearly independent and $m \leq n'$ ($n' \leq 2n + m$)
- The feasible set of a linear program forms a **polytope** (polyhedra bounded by faces of $n - 1$ dimension)
- The simplex algorithm moves between **vertices** of the polytope until it finds an optimal **vertex**
- Points on faces not perpendicular to $\boldsymbol{c}$ can be improved by sliding along the face in the direction of the projection of $-c$ onto the face.

Fundamental Theorem of LP

Theorem (Fundamental Theorem of Linear Programming)

Given:

$$\min\{\boldsymbol{c}^T\boldsymbol{x} \mid \boldsymbol{x} \in P\} \;\; \textit{where } P = \{\boldsymbol{x} \in \mathbb{R}^n \mid A\boldsymbol{x} \leq \boldsymbol{b}\}$$

If P is a bounded polyhedron and not empty and x^* is an optimal solution to the problem, then:

- $\boldsymbol{x}^*$ *is an extreme point (vertex) of P, or*
- $\boldsymbol{x}^*$ *lies on a face $F \subset P$ of optimal solution*

Proof:

- assume x^* not a vertex of P then $\exists$ a ball around it still in P. Show that a point in the ball has better cost
- if x^* is not a vertex then it is a convex combination of vertices. Show that all points are also optimal.

Simplex Algorithm

- Every vertex for a linear program in equality form can be uniquely defined by $n - m$ components of $\boldsymbol{x}$ that equal zero.
- choosing m design variables and setting the remaining variables to zero effectively removes $n - m$ columns of A, yielding an $m \times m$ constraint matrix
- the m selected columns of the matrix A are called **basis** and denoted by B: $x_i \geq 0$ for $i \in B$
- the $n - m$ columns not in B are called **not in basis** and are denoted by V: $x_i = 0$ for $i \in V$.

$$A\boldsymbol{x} = A_B \boldsymbol{x}_B = \boldsymbol{b} \quad \Longrightarrow \quad x_B = A_B^{-1} \boldsymbol{b}$$

Simplex Algorithm

- every vertex has an associated partition (B, V),
- not every partition corresponds to a vertex.
  A_B might be not invertible or the point $\boldsymbol{x}_B$ might not be ≥ 0.
- identifying partitions that correspond to vertices corresponds to solving an LP problem as well!

Two phases of the algorithm

1. **Initialization Phase**: finding a feasible starting vertex
2. **Optimization Phase**: finding the optimal vertex

Simplex Algorithm: FONCs

Lagrangian function:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) = \boldsymbol{c}^T \boldsymbol{x} - \mu^T \boldsymbol{x} - \lambda^T (A\boldsymbol{x} - \boldsymbol{b})$$

Conditions for Optimality for linear programs: KKT are also sufficient:

- feasibility: $A\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{x} \geq 0$
- dual feasibility: $\boldsymbol{\mu} \geq 0$
- complementary slackness: $\boldsymbol{\mu} \cdot \boldsymbol{x} = 0$
- stationarity: $A^T \boldsymbol{\lambda} + \boldsymbol{\mu} = \boldsymbol{c}$

$$A^T \boldsymbol{\lambda} + \boldsymbol{\mu} = \boldsymbol{c} \quad \Longrightarrow \quad \begin{cases} A_B^T \boldsymbol{\lambda} + \boldsymbol{\mu}_B = \boldsymbol{c}_B \\ A_V^T \boldsymbol{\lambda} + \boldsymbol{\mu}_V = \boldsymbol{c}_V \end{cases}$$

- We can choose $\mu_B = 0$ to satisfy complementry slackness (because $\boldsymbol{x}_B \geq 0$)

$$\boldsymbol{\mu}_V = \boldsymbol{c}_V - \left(A_B^{-1} A_V\right)^T \boldsymbol{c}_B$$

- Knowing $\boldsymbol{\mu}_V$ allows us to assess the optimality of the vertices. If $\boldsymbol{\mu}_B$ contains negative components, then dual feasibility is not satisfied and the vertex is sub-optimal.

- maintain a partition (B, V), which corresponds to a vertex of the feasible set polytope.

- The partition can be updated by swapping indices between B and V. Such a swap equates to moving from one vertex along an edge of the feasible set polytope to another vertex.

Simplex Algorithm: Optimization Phase

Pivoting

- $q \in V$ to enter in B

$$A\boldsymbol{x}' = A_B \boldsymbol{x}'_B + A_{\{q\}} x'_q = A_B \boldsymbol{x}_B = A\boldsymbol{x} = \boldsymbol{b}$$

- $p \in B$ to leave B becomes zero during the transition.

$$\boldsymbol{x}'_B = \boldsymbol{x}_B - A_B^{-1} A_{\{q\}} x'_q \quad \Longrightarrow \quad (\boldsymbol{x}'_B)_p = 0 = (\boldsymbol{x}_B)_p - \left(A_B^{-1} A_{\{q\}}\right)_p x'_q$$

- leaving index is obtained using the **minimum ratio test**: compute x'_q for each potential leaving index p and select the leaving index p that yields the smallest x'_q.

- Choosing an entering index q decreases the objective function value by

$$\boldsymbol{c}^T \boldsymbol{x}' = \boldsymbol{c}_B^T \boldsymbol{x}'_B + c_q x'_q = \boldsymbol{c}^T \boldsymbol{x} + \mu_q \boldsymbol{x}'_q$$

- The objective function decreases only if μ_q is negative.

Simplex Algorithm: Optimization Phase

- In order to progress toward optimality, we must choose an index q in V such that μ_q is negative. If all components of μ_V are non-negative, we have found a global optimum.
- Since there can be multiple negative entries in μ_V, Several possible heuristics to search for optimal vertex (choose next q)
  - **Dantzig's rule**: choose most negative entry in μ_V; easy to calculate
  - **Greedy heuristic (largest decrease)**: maximally reduces objective at each step
  - **Bland's rule**: chooses first vertex found with negative μ_V; useful for preventing or breaking out of cycles

Simplex Algorithm: Initialization Phase

- The starting vertex of the optimization phase is found by solving an additional **auxiliary linear program** that has a known feasible starting vertex

$$\begin{aligned} \underset{\boldsymbol{x},\boldsymbol{z}}{\text{minimize}} \quad & \begin{bmatrix} 0^T & 1^T \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{z} \end{bmatrix} \\ & \begin{bmatrix} A & Z \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{z} \end{bmatrix} = \boldsymbol{b} \\ & \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{z} \end{bmatrix} \geq 0 \end{aligned}$$

- The solution is a feasible vertex in the original linear program

Dual Certificates

- Verification that the solution returned by the algorithm is actually the correct solution
- Recall that the solution to the dual problem, d^* provides a lower bound to the solution of the primal problem, p^*
- If $d^* = p^*$ then p^* is guaranteed to be the unique optimal value because the duality gap is zero
- What happens if one of the two is unbounded or infeasible?

Dual Certificates

Linear programs have a simple dual form:

Primal form (equality)

$$\begin{aligned} \underset{\boldsymbol{x}}{\text{minimize}} \quad & \boldsymbol{c}^T \boldsymbol{x} \\ \text{subject to} \quad & A\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geq 0 \end{aligned}$$

Dual form

$$\begin{aligned} \underset{\boldsymbol{\lambda}}{\text{maximize}} \quad & \boldsymbol{b}^T \boldsymbol{\lambda} \\ \text{subject to} \quad & A^T \boldsymbol{\lambda} \leq \boldsymbol{c} \end{aligned}$$

Strong Duality Theorem

Due to Von Neumann and Dantzig 1947 and Gale, Kuhn and Tucker 1951.

Theorem (Strong Duality Theorem)

Given:

$$\begin{aligned} &(P)\ \min\{\boldsymbol{c}^T\boldsymbol{x} \mid A\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{x} \geq 0\} \\ &(D)\ \max\{\boldsymbol{b}^T\boldsymbol{\lambda} \mid A^T\boldsymbol{\lambda} \geq \boldsymbol{c}\} \end{aligned}$$

exactly one of the following occurs:

1. *(P) and (D) are both infeasible*
2. *(P) is unbounded and (D) is infeasible*
3. *(P) is infeasible and (D) is unbounded*
4. *(P) has feasible solution, then let an optimal be:* $\boldsymbol{x}^* = [x_1^*, \ldots, x_n^*]$
   (D) has feasible solution, then let an optimal be: $\boldsymbol{\lambda}^* = [\lambda_1^*, \ldots, \lambda_m^*]$*, then:*

$$p^* = \boldsymbol{c}^T\boldsymbol{x}^* = \boldsymbol{b}^T\boldsymbol{\lambda}^* = d^*$$

Summary

- Linear programs are problems consisting of a linear objective function and linear constraints
- The simplex algorithm can optimize linear programs globally in an efficient manner
- Dual certificates allow us to verify that a candidate primal-dual solution pair is optimal
- Linear programs can be solved to optimality for problems with millions of variables.

15. Sampling Plans

Sampling Plans

- In all nonlinear non convex optimization, to generate good initial design points
- With computationally costly functions, to create an initial set of design points from where to build a **surrogate models** to optimize in place of the original function
- In hyperparameter tuning

Full Factorial Design

- Factors and levels, terms from the field of Experimental Design in Statistics
- Uniform and evenly spaced samples across domain
- Simple, easy to implement, and covers domain
- Optimization over the points known as **grid search**
- Sample count grows exponentially with dimension: n^m
- Can be coarse and miss local features

$a_i \leq x_i \leq b_i$ for each component i.
grid with m_i samples in the ith dimension

Random Sampling

- Uses pseudorandom number generator to define samples according to our desired distribution
- If variable bounds are known, a common choice is independent **uniform distributions** across domains of possible variable values
  $[a_1, b_1] \times \ldots \times [a_n, b_n]$
- Ideally, if enough points are sampled and the right distribution is chosen, the design space will be covered

Uniform Projection Plans

- A **uniform projection plan** is a sampling plan over a discrete grid where the distribution over each dimension is uniform.

Example

In 2D, $m \times m$ sampling grid (as in full factorial), but, instead of taking all m^2 samples, we want to sample only m positions.

Uniform Projection Plans

Example (Random m-permutations)

$p = 4\ 2\ 1\ 3\ 5$

Example (Latin square)

Latin squares are $m \times m$ grids where each row contains each integer 1 through m and each column contains each integer 1 through m.

Latin-hypercubes are a generalization to any number of dimensions (note that the points remain m) N rooks on a chess board without threatening each other

4	1	3	2
1	4	2	3
3	2	1	4
2	3	4	1

Stratified Sampling

- Each point is sampled uniformly at random within each grid cell instead of the center
- Cells decided by Full Factorial or Uniform Projection Plans
- Can capture details that regularly-spaced samples might miss

Space Filling Metrics

- A sampling plan may cover a search space fully, but still leave large areas unexplored

Example (Uniform Projection Plan)

- **space-filling metrics** quantify this aspect measuring the degree to which a sampling plan $X \subseteq \mathcal{X}$ fills the design space

Space-Filling Metrics: Discrepancy

- **Discrepancy**: measure of ability of the sampling plan X to fill a hyper-rectangular design space

- It is given by hyper-rectangular subset $\mathcal{H}$ with the maximum difference between the fraction of samples in $\mathcal{H}$ and the volume of $\mathcal{H}$'s.

$$d(X) = \underset{\mathcal{H}}{\text{supremum}} \left| \frac{\#(X \cap \mathcal{H})}{\#X} - \lambda(\mathcal{H}) \right|$$

$\lambda(\mathcal{H})$ is the n-dimensional volume of $\mathcal{H}$, ie, the product of the side lengths of $\mathcal{H}$

We wish to have a plan X with low discrepancy

Often very difficult to compute directly

d for the purple rectangle is $>$ than d for the blue rectangle

Space-Filling Metrics: Pairwise Distances

- Method of measuring relative space-filling performance of two m-point sampling plans
- Better spread-out plans will have larger pairwise distances:
  1. compute all pairwise distances between all points within each sampling plan
  2. sort the pairwise distances of each set in ascending order
  3. the plan with the first pairwise distance exceeding the other is considered more space-filling
- Suggests simple algorithm:
  1. produce a set of randomly distributed sampling plans,
  2. pick the one with greatest pairwise distances
- Possible also for uniform projection plans, by mutating them with swaps and simulated annealing.

Space-Filling Metrics: Morris-Mitchell Criterion

- Alternative to previously suggested algorithm that simplifies the optimization problem

$$\underset{X}{\text{minimize}} \quad \underset{q \in \{1,2,3,10,20,50,100\}}{\text{maximize}} \quad \Phi_q(X)$$

$$\Phi_q(X) = \left(\sum_i d_i^{-q} \right)^{\frac{1}{q}}$$

where d_i is the ith pairwise distance between points in X and $q > 0$ is a tunable parameter. Larger values of q give higher penalties to large distances.

Space-Filling Metrics: Morris-Mitchell Criterion

Uniform projection plans sorted from best to worst according to Φ_1

Space-Filling Subsets

- Often, the set of possible sample points is constrained to be a subset of available choices
- A space-filling metric for a subset S within a finite set X is the maximum distance between a point in X and the closest point in S, using a norm to measure distance

$$d_{\max}(X, S) = \underset{\boldsymbol{x} \in X}{\text{maximize}} \; \underset{\boldsymbol{s} \in S}{\text{minimize}} \; \|\boldsymbol{s} - \boldsymbol{x}\|_q$$

- A space-filling subset minimizes this metric
- Often computationally intractable, but heuristics like (repeated) greedy construction and exchange-search often produce acceptable results

Space-Filling Subsets

Quasi-Random Sequences

- Also called **low-discrepancy sequences**, **quasi-random sequences** are deterministic sequences that systematically fill a space such that their integral over the space converges as fast as possible

- Used for fast convergence in Monte Carlo integration, which approximates an integral by sampling points in a domain

- Quasi-random sequences are typically constructed for the unit n-dimensional hypercube, $[0, 1]^n$. Any multidimensional function with bounds on each variable can be transformed into such a hypercube.

Quasi-Random Sequences

- Additive Recurrence: Recursively adds irrational numbers
- Halton Sequence: sequence of fractions generated with coprime numbers
- Sobol Sequence: recursive XOR operation with carefully chosen numbers

Quasi-Random Sequences: Additive Recurrence

- Recursively adds irrational numbers

$$x_{k+1} = x_k + c \pmod 1$$

c irrational

$$c = 1 - \varphi = \frac{\sqrt{5} - 1}{2} \approx 0.618034$$

φ is golden ratio

- We can construct a space-filling set over n dimensions using an additive recurrence sequence for each coordinate, each with its own value of c.

- square roots of the primes are known to be irrational, and can thus be used to obtain different sequences for each coordinate:

$$c_1 = \sqrt{2}, c_2 = \sqrt{3}, c_3 = \sqrt{5}, c_4 = \sqrt{7}, c_5 = \sqrt{11}, \ldots$$

Quasi-Random Sequences: Halton Sequence

- single-dimensional version, called van der Corput sequences, generates sequences where the unit interval is divided into powers of base b. For example, $b = 2$

$$X = \left\{ \frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{5}{8}, \frac{3}{8}, \frac{7}{8}, \frac{1}{16}, \ldots \right\}$$

  whereas $b = 5$

$$X = \left\{ \frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}, \frac{1}{25}, \frac{6}{25}, \frac{11}{25}, \ldots \right\}$$

- Multi-dimensional space-filling sequences use one van der Corput sequence for each dimension, each with its own base b. The bases, however, must be **coprime** in order to be uncorrelated.

- Two integers are **coprime** if the only positive integer that divides them both is 1, eg, 8 and 9.

- Correlation can be avoided by the leaped Halton method, which takes every pth point, where p is a prime different from all coordinate bases.

Quasi-Random Sequences: Sobol Sequence

- Recursive XOR operation with carefully chosen numbers.
- XOR ($\oplus$) returns true if and only if both inputs are different
- For n-dimensional hypercube $I^n = [0,1]^n$, the ith point of the sequence $\boldsymbol{x}_i$ for dimension j is calculated as:

  $$x_{i,j} = x_{i-1,j} \oplus v_{k,j}$$

  $v_{k,j}$ is the jth dimension of the kth direction number.
- **direction numbers** $v_{k,j} = (0.v_{k,j,1}v_{k,j,2}\ldots)_2$ where $v_{k,j,m}$ denotes the mth digit after the binary point.
- Tables of direction numbers with different properties have been proposed.
- Initialization: unit initialisation: ℓth left most bit set to one $v_{k,j,\ell} = 1$ for all k and j and all others to be zero

Quasi-Random Sequences

space-filling sampling plans in two dimensions. Samples are colored according to the order in which they are sampled. The uniform projection plan was generated randomly and is not optimized.

Summary

- Sampling plans are used to cover search spaces with a limited number of points
- Full factorial sampling, which involves sampling at the vertices of a uniformly discretized grid, requires a number of points exponential in the number of dimensions
- Uniform projection plans, which project uniformly over each dimension, can be efficiently generated and can be optimized to be space-filling
- Greedy construction and the exchange local search algorithm can be used to find a subset of points that maximally fill a space
- Quasi-random sequences are deterministic procedures by which space-filling sampling plans can be generated

16. Discrete Optimization: Integer Linear Programming, Dynamic Programming

Discrete Optimization

- Discrete optimization is a branch of optimization that deals with problems where the solution space is discrete, meaning that the variables can only take on specific, distinct values.
- This is in contrast to continuous optimization, where the variables can take on any value within a given range.
- Discrete optimization problems are often NP-hard, meaning that they are computationally challenging to solve.
- Discrete optimization is widely used in various fields, including operations research, computer science, engineering, and economics.
- It is important to develop efficient algorithms and heuristics to solve these problems, as they often arise in real-world applications.

Discrete Optimization

- What is discrete optimization?
- Problem formulation for linear programs
- Approximate solution techniques
- Exact solution techniques
- Dynamic programming

Discrete vs Combinatorial Optimization

In Combinatorial Optimization, variables have some combinatorial structure, ie, sets, permuations, paths, etc.

Definition (Combinatorial Optimization Problem (COP))

Input: Given a finite set $N = \{1, \ldots, n\}$ of objects,
weights c_j for all $j \in N$,
a collection of feasible subsets of N, $\mathcal{F}$
Task: Find a minimum weight feasible subset, ie,

$$\underset{S \subseteq N}{\text{minimize}} \left\{ \sum_{j \in S} c_j \mid S \in \mathcal{F} \right\}$$

COP can also be modelled as discrete optimization problems.

Typically: **incidence vector** of S, $\boldsymbol{x}^S \in \mathbb{B}^n$: $x_j^S = \begin{cases} 1 & \text{if } j \in S \\ 0 & \text{otherwise} \end{cases}$

Discrete (or Combinatorial) Optimization

Solving problems with variables that are discrete instead of continuous

Example

- Set covering and set partitioning
- Knapsack problem
- Traveling salesman problem
- Vehicle routing problem
- Job scheduling problem
- Facility location problem
- Bin packing problem
- Graph coloring problem
- Maximum flow problem
- Minimum spanning tree problem
- Shortest path problem
- Steiner tree problem
- Hamiltonian path problem

The set of possible discrete values can be finite or infinite

Solution Methods

Exact Methods:

- Integer programming
- Combinatorial optimization algorithms
- Graph theory algorithms
- Scheduling algorithms
- SAT
- Dedicated Branch and bound
- Dynamic programming
- Constraint programming
- Integer linear programming
- Mixed-integer programming
- Network flow algorithms

Heuristic Methods:

- Approximation algorithms
- Greedy algorithms
- Metaheuristics
  - Local Search algorithms
  - Genetic algorithms
  - Simulated annealing
  - Ant colony optimization
  - Tabu search

Outline

18. Integer Linear Programming

19. Dynamic Programming

Notation: Set of Integer Numbers

- $\mathbb{Z}$ set of integer numbers $\{..., -3, -2, -1, 0, 1, 2, 3, ...\}$
- $\mathbb{Z}^+$ set of positive integers
- $\mathbb{Z}_0^+$ set of nonnegative integers ($\{0\} \cup \mathbb{Z}^+$)
- $\mathbb{N}_0$ set of natural numbers, ie, nonnegative integers $\{0, 1, 2, 3, 4, ...\}$
- $\mathbb{B}$ set of binary numbers

Mixed Integer Linear Programming (ILP)

Linear Objective • Linear Constraints • but! integer variables

$$\begin{aligned} \max\ & \boldsymbol{c}^T\boldsymbol{x} \\ & A\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \geq 0 \end{aligned}$$

Linear Programming (LP)

$$\begin{aligned} \max\ & \boldsymbol{c}^T\boldsymbol{x} \\ & A\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \quad \text{integer} \end{aligned}$$

Integer Linear Programming (ILP)

$$\begin{aligned} \max\ & \boldsymbol{c}^T\boldsymbol{x} \\ & A\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \in \{0,1\}^n \end{aligned}$$

Binary Integer Program (BIP)
0/1 Integer Programming

$$\begin{aligned} \max\ & \boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{h}^T\boldsymbol{y} \\ & A\boldsymbol{x} + G\boldsymbol{y} \leq \boldsymbol{b} \\ & \boldsymbol{x} \geq 0 \\ & \boldsymbol{y} \geq 0 \\ & \boldsymbol{y} \quad \text{integer} \end{aligned}$$

Mixed Integer Linear Programming (MILP)

$$\begin{aligned} \max\ & f(\boldsymbol{x}) \\ & g(\boldsymbol{x}) \leq \boldsymbol{b} \\ & \boldsymbol{x} \quad \text{integer} \end{aligned}$$

Integer Non-linear Programming (INLP)

Mathematical Programming: Modeling

- Find out exactly what the decision maker needs to know:
  - which investment?
  - which product mix?
  - which job j should a person i do?
- Define **Decision Variables** of suitable type (continuous, integer valued, binary) corresponding to the needs and **Known Parameters** corresponding to given data.
- Formulate **Objective Function** computing the benefit/cost
- Formulate mathematical **Constraints** indicating the interplay between the different variables.

Rounding

$$\begin{array}{rcrcl} \max\ 100x_1 & + & 64x_2 & & \\ 50x_1 & + & 31x_2 & \leq & 250 \\ 3x_1 & - & 2x_2 & \geq & -4 \\ & & x_1, x_2 & \in & \mathbb{Z}_0^+ \end{array}$$

LP optimum $(376/193, 950/193)$
IP optimum $(5, 0)$

Note: rounding does not help in the example above!

$\rightsquigarrow$ feasible region convex but not continuous: Now the optimum can be on the border (vertices) but also **internal**.

Possible way: solve the **relaxed** problem.

- If solution is **integer**, done.
- If solution is **rational** (never irrational) try rounding to the nearest integers (but may exit feasibility region)
  if in $\mathbb{R}^2$ then 2^2 possible roundings (up or down)
  if in $\mathbb{R}^n$ then 2^n possible roundings (up or down)

Rounding

Rounding to infeasible point

Rounding to suboptmal point

If A is integral, the error of a rounded solution can be bounded

Cutting Planes

$$
\begin{array}{rrcl}
\max & x_1 + 4x_2 & & \\
 & x_1 + 6x_2 & \leq & 18 \\
 & x_1 & \leq & 3 \\
 & x_1, x_2 & \geq & 0 \\
 & x_1, x_2 \text{ integers} & &
\end{array}
$$

Cuts: Useful valid inequalities: they are satisfied by all integer solutions but not by all relaxed solutions

Cutting Plane Method

- The cutting plane method solves the relaxed LP, adds linear constraints, then repeats until the solution is exact
- Solves mixed integer programs exactly
- The linear constraints are chosen such that all discrete points are still feasible, but the relaxed solution is not

Chvatal-Gomory's Cutting Plane Algorithm

- Recall that we can partition a vertex (and also an optimal one $\boldsymbol{x}^*$) as

$$A_B \boldsymbol{x}_B^* + A_N \boldsymbol{x}_N^* = \boldsymbol{b}$$

- Using the method of Gomory's cut, we can add an additional inequality constraint for each nonintegral dimension

$$x_b^* - \lfloor x_b^* \rfloor - \sum_{j \in N} \left(\bar{A}_{bj} - \lfloor \bar{A}_{bj} \rfloor \right) x_j \leq 0 \qquad \bar{A} = A_B^{-1} A_N$$

- This "cuts out" the relaxed solution $\boldsymbol{x}^*$

$$\underbrace{x_b^* - \lfloor x_b^* \rfloor}_{>0} - \underbrace{\sum_{j \in N} \left(\bar{A}_{bj} - \lfloor \bar{A}_{bj} \rfloor \right) x_j^*}_{=0} > 0$$

Branch and Bound

- Algorithm for efficiently searching the very large set of solution possibilities (first proposed by Ailsa Land and Alison Doig, "An automatic method of solving discrete programming problems", 1960)
- **Branching** is dividing the domain into sections
- **Bounding** is keeping track of the best solution so far and rejecting regions that cannot improve upon it
- In the worst case, the algorithm has to search all possibilities but in practice it works very well. Combined with cutting planes, this approach forms the basis of many commercial MIP solvers.

Branch and Bound

Example

$$
\begin{array}{rrcrcl}
\max & x_1 & + & 2x_2 & & \\
 & x_1 & + & 4x_2 & \leq & 8 \\
 & 4x_1 & + & x_2 & \leq & 8 \\
 & & & x_1, x_2 & \geq & 0, \text{integer}
\end{array}
$$

4.8
$x_1 \leq 1$
$x_1 \geq 2$

x_2
$x_1 = 1$
$x_1 + 4x_2 = 8$
x_1
$x_1 + 2x_2 = 1$
$4x_1 + x_2 = 8$

x_2
$x_1 + 4x_2 = 8$
x_1
$x_1 + 2x_2 = 1$
$4x_1 + x_2 = 8$

4.8
$-\infty$
$x_1 \leq 1$
$x_1 \geq 2$
4.5
$-\infty$
2
2
x_1=2
x_2=0
$x_2 \leq 1$
$x_2 \geq 2$
x_1=1
x_2=1
3
3
4
4
x_1=0
x_2=2

x_2
$x_1 + 4x_2 = 8$
x_1
$x_1 + 2x_2 = 1$
$4x_1 + x_2 = 8$

x_2
$x_1 + 4x_2 = 8$
x_1
$x_1 + 2x_2 = 1$
$4x_1 + x_2 = 8$

Branch and Bound: Pruning

$$\underset{x,y}{\text{minimize}} \; -1.3x - y$$

$$1.8x + y \leq 7.$$

$$y \leq 5$$

$$x, y \in \mathbb{Z}$$

Pruning by: integrality, bounding, infeasibiliy.

Outline

18. Integer Linear Programming

19. Dynamic Programming

Dynamic Programming

- Applied to problems with **optimal substructure** and **overlapping subproblems**
- Optimal substructure means an optimal solution can be constructed from optimal solutions to its subproblems
- Overlapping subproblems means solving each subproblem separately requires repeating certain operations
- **Dynamic programming** begins with desired problem and recurses down to smaller and smaller subproblems, retrieving the value of previously solved problems as necessary
- Principle of Optimality (known as Bellman Optimality Conditions): Suppose that the solution of a problem is the result of a sequence of n decisions $D_1, D_2, ..., D_n$; if a given sequence is optimal, then the first k decisions must be optimal, but also the last $n - k$ decisions must be optimal
- DP breaks down the problem into stages, at which decisions take place, and find a recurrence relation that relates each stage with the previous one

Example 1: Knapsack Problem

- Trying to pack some number of items into a backpack
- Limited space in the backpack
- Each item has a specified value and size
- What is the best subset of items to include?

A MILP Formulation

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & -\sum_{i=1}^{n} v_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \leq W \\ & x_i \in \{0,1\} \text{for } i = 1, 2, \ldots, n \end{aligned}$$

Dynamic Programming

- Let $\text{knapsack}(i, w)$ be the maximum value achievable using the first i items and a knapsack capacity w.
- Consider the ith item. You can either use it or not.
- If you don't use it, then the value of your knapsack will be

$$\text{knapsack}(i-1, w)$$

- If you use it, then the value of your knapsack will be

$$\text{knapsack}(i-1, w-w_i) + v_i$$

The Recursion

- For each item i and weight w:

$$\text{knapsack}(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ \max \begin{cases} \text{knapsack}(i-1, w) & \text{(discard new item)} \\ \text{knapsack}(i-1, w - w_i) + v_i & \text{(include new item)} \end{cases} & \text{if } w_i \leq w \\ \text{knapsack}(i-1, w) & \text{if } w_i > w \end{cases}$$

- Optimal solution:

$$z^* = \text{knapsack}(n, W)$$

and trace back to find the items collected.

Example

Capacity = 5 kg

Item	Weight	Value
1	1 kg	\$1
2	3 kg	\$4
3	4 kg	\$5

Step 1: Initialize

Create a table with (number of items + 1) rows and (capacity + 1) columns, filled with zeros.

```
  w = 0 1 2 3 4 5
i=0 | 0 0 0 0 0 0
i=1 | 0
i=2 | 0
i=3 | 0
```

Step 2: Fill it step by step

```
Item 1 (w_1=1, v_1=1):
  w = 0 1 2 3 4 5
i=0 | 0 0 0 0 0 0
i=1 | 0 1 1 1 1 1
i=2 | 0
i=3 | 0
```

```
Item 2 (w_2=3, v_2=4):
  w = 0 1 2 3 4 5
i=0 | 0 0 0 0 0 0
i=1 | 0 1 1 1 1 1
i=2 | 0 1 1 4 5 5
i=3 | 0

Item 3 (w_3=4, v_3=5):
  w = 0 1 2 3 4 5
i=0 | 0 0 0 0 0 0
i=1 | 0 1 1 1 1 1
i=2 | 0 1 1 4 5 5
i=3 | 0 1 1 4 5 6
```

Example 2: Padovan Sequence

$$P_n = P_{n-2} + P_{n-3} \qquad P_0 = P_1 = P_2 = 1$$

```
def padovan_naive(n, P=None):
    if n < 3:
        return 1
    else:
        return padovan_naive(n - 2, P) +↪
            ↪ padovan_naive(n - 3, P)
```

P_9
P_7 P_6
P_5 P_4 P_4 P_3
P_3 P_2 P_2 P_1 P_2 P_1 P_1 P_0
P_1 P_0

Timing Naive Padovan Sequence
Elapsed Time
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
0
10
20
30
40
n

```python
def padovan_topdown(n, P=None):
    if P is None:
        P = {}
    if n not in P:
        if n < 3:
            P[n] = 1
        else:
            P[n] = padovan_topdown(n - 2, P) + ↪
                ↪padovan_topdown(n - 3, P)
    return P[n]

def padovan_bottomup(n):
    P = {0: 1, 1: 1, 2: 1}
    for i in range(3, n + 1):
        P[i] = P[i - 2] + P[i - 3]
    return P[n]
```

```python
from functools import lru_cache

@lru_cache(maxsize=None)
def padovan_topdown(n):
    if n < 3:
        return 1
    return padovan_topdown(n - 2) + padovan_topdown(n - 3)

def padovan_bottomup_const(n):
    if n < 3:
        return 1
    p0, p1, p2 = 1, 1, 1  # Corresponds to P[0], P[1], P[2]
    for _ in range(3, n + 1):
        p_next = p0 + p1
        p0, p1, p2 = p1, p2, p_next
    return p2
```

Padovan Sequence Runtime Comparison
Top-Down with lru_cache
Bottom-Up with dict
Bottom-Up Constant Space
Time (seconds)
0.000
0.001
0.002
0.003
0.004
0.005
0
2000
4000
6000
8000
10000
n

Example 3: Traveling Salesman Problem

https://www.math.uwaterloo.ca/tsp/

Principle of Optimality

The TSP asks for the shortest tour that starts from 0, visits all cities of the set $C = \{1, 2, ..., n\}$ exactly once, and returns to 0, where the cost to travel from i to j is c_{ij} (with $(i, j) \in A$)
If the optimal solution of a TSP with six cities is $(0, 1, 3, 2, 4, 6, 5, 0)$, then...

- the optimal solution to visit $\{1, 2, 3, 4, 5, 6\}$ starting from 0 and ending at 5 is $(0, 1, 3, 2, 4, 6, 5)$
- the optimal solution to visit $\{1, 2, 3, 4, 6\}$ starting from 0 and ending at 6 is $(0, 1, 3, 2, 4, 6)$
- the optimal solution to visit $\{1, 2, 3, 4\}$ starting from 0 and ending at 4 is $(0, 1, 3, 2, 4)$
- the optimal solution to visit $\{1, 2, 3\}$ starting from 0 and ending at 2 is $(0, 1, 3, 2)$
- the optimal solution to visit $\{1, 3\}$ starting from 0 and ending at 3 is $(0, 1, 3)$
- the optimal solution to visit 1 starting from 0 is $(0, 1)$

$\rightsquigarrow$ The optimal solution is made up of a number of optimal solutions of smaller subproblems

Enumerate All Solutions of the TSP

- A solution of a TSP with n cities derives from a sequence of n decisions, where the kth decision consists of choosing the kth city to visit in the tour

- The number of nodes (or states) grows exponentially with n
- At stage k, the number of states is $\binom{n}{k}k!$
- With $n = 6$, at stage $k = 6$, 720 states are necessary

$\rightsquigarrow$ DP finds the optimal solution by implicitly enumerating all states but actually generating only some of them

Are All States Necessary?

If path $(0, 1, 2, 3)$ costs less than $(0, 2, 1, 3)$, the optimal solution cannot be found in the blue part of the tree

Are All States Necessary?

If path $(0, 1, 2, 3, 4, 5)$ costs less than $(0, 1, 2, 4, 3, 5)$, the optimal solution cannot be found in the blue part of the tree

Are All States Necessary?

- At stage k $(1 \leq k \leq n)$, for each subset of cities $S \subseteq C$ of cardinality k, it is necessary to have only k states (one for each of the cities of the set S)
- At state $k = 3$, given the subset of cities $S = \{1, 2, 3\}$, three states are needed:
  - the shortest-path to visit S by starting from 0 and ending at 1
  - the shortest-path to visit S by starting from 0 and ending at 2
  - the shortest-path to visit S by starting from 0 and ending at 3
- At stage k, $\binom{n}{k} k$ states are required to compute the optimal solution (not $\binom{n}{k} k!$)

#States n = 6

Stage	$\binom{n}{k} k!$	$\binom{n}{k} k$
1	6	6
2	30	30
3	120	60
4	360	60
5	720	30
6	720	6

Complete Trees with n=4

Dynamic Programming Recursion for the TSP I

- Given a subset $S \subseteq C$ of cities and $k \in S$, let $f(S, k)$ be the optimal cost of starting from 0, visiting all cities in S, and ending at k
- Begin by finding $f(S, k)$ for $|S| = 1$, which is $f(\{k\}, k) = c_{0k}, \forall k \in C$
- To compute $f(S, k)$ for $|S| > 1$, the best way to visit all cities of S by starting from 0 and ending at k is to consider all $j \in S \setminus \{k\}$ immediately before k, and look up $f(S \setminus \{k\}, j)$, namely

$$f(S, k) = \min_{j \in S \setminus \{k\}} \{f(S \setminus \{k\}, j) + c_{jk}\}$$

- The optimal solution cost z^* of the TSP is $z^* = \min_{k \in C}\{f(C, k) + c_{k0}\}$

Dynamic Programming Recursion for the TSP II

DP Recursion from [Held and Karp (1962)]

1. **Initialization**. Set $f(\{k\}, k) = c_{0k}$ for each $k \in C$

2. **RecursiveStep**. For each stage $r = 2, 3, ..., n$, compute

$$f(S, k) = \min_{j \in S \setminus \{k\}} \{f(S \setminus \{k\}, j) + c_{jk}\} \forall S \subseteq C : |S| = r \text{ and } \forall k \in S$$

3. **Optimal Solution**. Find the optimal solution cost z^* as

$$z^* = \min_{k \in C} \{f(C, k) + c_{k0}\}$$

- With the DP recursion, TSP instances with up to 25 - 30 customers can be solved to optimality; other solution techniques (i.e., branch-and-cut) are able to solve TSP instances with up to... 85900 customers
- Nonetheless, DP recursions represents the state-of-the-art solution techniques to solve a wide variety of PDPs

TSP

Thiago Serra · 2nd
Assistant Professor of Business Analytics at University of Iowa
4d ·
+ Follow
Bill Cook strikes again: shortest tour of 81,998 bars in South Korea is the top story tonight at Hacker News
Link: https://lnkd.in/djSAkt5E
T-Mobile
11:30 PM
Hacker News
new | past | comments | ask | show | jobs | submit
login
1. Shortest-possible walking tour to 81,998 bars in South Korea (uwaterloo.ca)
150 points by geeknews 4 hours ago | hide | 32 comments
2. Show HN: My from-scratch OS kernel that runs DOOM (github.com/unmappedstack)
100 points by UnmappedStack 3 hours ago | hide | 19 comments
3. How a 20 year old bug in GTA San Andreas surfaced in Windows 11 24H2
(cookieplmonster.github.io)

Summary

- Discrete optimization problems require that the design variables be chosen from discrete sets.
- Relaxation, in which the continuous version of the discrete problem is solved, is by itself an unreliable technique for finding an optimal discrete solution but is central to more sophisticated algorithms.
- Many combinatorial optimization problems can be framed as an integer program, which is a linear program with integer constraints.
- Both the cutting plane and branch and bound methods can be used to solve integer programs efficiently and exactly. The branch and bound method is quite general and can be applied to a wide variety of discrete optimization problems.
- Dynamic programming is a powerful technique that exploits optimal overlapping substructure in some problems.

17. Discrete Optimization: Constraint Programming and Heuristics

Outline

20. Constraint Programming

21. Randomized Optimization Heuristics

Number Circle Puzzle

You have 8 minutes

Put a different number (1 to 8) in each circle such that adjacent circles do not have consecutive numbers.

Example by Patrick Prosser with the help of Toby Walsh, Chris Beck, Barbara Smith, Peter van Beek, Edward Tsang.

Heuristic Search

Which nodes are hardest to number?

Heuristic Search

Heuristic Search

Which are the least constraining values to use?

Heuristic Search

Values 1 and 8

Heuristic Search

Values 1 and 8

Symmetry means we don't need to consider: 8 1

Inference/propagation

We can now eliminate many values for other nodes

Inference/propagation

{1,2,3,4,5,6,7,8}

Inference/propagation

Inference/propagation

Inference/propagation

By symmetry

Inference/propagation

Inference/propagation

Inference/propagation

Inference/propagation

By symmetry

Inference/propagation

Inference/propagation

Value 2 and 7 are left in just one variable domain each

Inference/propagation

And propagate …

Inference/propagation

And propagate …

Inference/propagation

And propagate …

Inference/propagation

Guess a value, but be prepared to backtrack …

Inference/propagation

Guess a value, but be prepared to backtrack …

Inference/propagation

And propagate …

Inference/propagation

And propagate …

Inference/propagation

Guess another value …

Inference/propagation

Guess another value …

Inference/propagation

And propagate …

Inference/propagation

And propagate …

Inference/propagation

One node has only a single value left …

Inference/propagation

{6}

Solution

The Core of Constraint Computation

- Modelling
  - Deciding on variables/domains/constraints
- Heuristic Search
- Inference/Propagation
- Symmetry
- Backtracking

Hardness

- The puzzle is actually a hard problem
  - NP-complete

Constraint programming

- **Model** problem by specifying constraints on acceptable solutions:
  - define variables and domains
  - post constraints on these variables
- Solve model
  - choose algorithm
    - incremental assignment / backtracking search
    - complete assignments / stochastic search
  - design heuristics

Constraint Satisfaction Problem

- Variable x_i for each node $i = 1, \ldots, 8$
- Domain $\{1, \ldots, 8\}$ for each variable x_i
- Constraints:

$$\texttt{allDifferent}([x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8])$$
$$|x_1 - x_2| > 1$$
$$|x_2 - x_3| > 1$$
$$|x_3 - x_4| > 1$$
$$\vdots$$
$$|x_7 - x_8| > 1$$

Modeling in Constraint Programming

The **domain** of a variable x, denoted $D(x)$, is a finite set of elements that can be assigned to x.

A **constraint** C on X is a subset of the Cartesian product of the domains of the variables in X, i.e., $C \subseteq D(x_1) \times \cdots \times D(x_k)$. A tuple $(d_1, \ldots, d_k) \in C$ is called a **solution** to C.

Equivalently, we say that a solution $(d_1, ..., d_k) \in C$ is an assignment of the value d_i to the variable x_i for all $1 \leq i \leq k$, and that this assignment satisfies C.

If $C = \emptyset$, we say that it is **inconsistent**.

Modeling in Constraint Programming

Constraint Satisfaction Problem (CSP)

A CSP is a finite set of variables X with **domain extension** $\mathcal{D} = D(x_1) \times \cdots \times D(x_n)$, together with a finite set of constraints $\mathcal{C}$, each on a subset of X. A **solution** to a CSP is an assignment of a value $d \in D(x)$ to each $x \in X$, such that all constraints are satisfied simultaneously.

Constraint Optimization Problem (COP)

A COP is a CSP $\mathcal{P}$ defined on the variables $x_1, \ldots, x_n$, together with an objective function $f : D(x_1) \times \cdots \times D(x_n) \to Q$ that assigns a value to each assignment of values to the variables. An **optimal solution** to a minimization (maximization) COP is a solution d to $\mathcal{P}$ that minimizes (maximizes) the value of $f(d)$.

Another Example: Social Golfers

Example (Social Golfer Problem (Combinatorial Design))

- 9 golfers: 1, 2, 3, 4, 5, 6, 7, 8, 9
- wish to play in groups of 3 players in 4 days
- such that no golfer plays in the same group with any other golfer more than just once.

Is it possible?

	Group 1	**Group 2**	**Group 3**
Day 0	? ? ?	? ? ?	? ? ?
Day 1	? ? ?	? ? ?	? ? ?
Day 2	? ? ?	? ? ?	? ? ?
Day 3	? ? ?	? ? ?	? ? ?

This is an instance of a **constrained satisfaction problem**. Adding an optimizing criterion we get a **constrained optimization problem**.

Solution Paradigms

1. Dedicated algorithms
   (eg.: enumeration, branch and bound, dynamic programming)
2. Constraint Programming
3. Integer Linear Programming
4. Other modeling (SAT, SMT, etc.)
5. Randomized Search/Optimization Heuristics

Common to 2-5: **Representation (modeling) + reasoning (search + inference)**

Constraint Programming: Representation

Golfers

	Group 1	**Group 2**	**Group 3**
Day 0	? ? ?	? ? ?	? ? ?
Day 1	? ? ?	? ? ?	? ? ?
Day 2	? ? ?	? ? ?	? ? ?
Day 3	? ? ?	? ? ?	? ? ?

Groups

	Day 0	Day 1	Day 2	Day 3
Golfer 0	1	{1,2,3}	{1,2,3}	{1,2,3}
Golfer 1	1	{1,2,3}	{1,2,3}	{1,2,3}
Golfer 2	1	{1,2,3}	{1,2,3}	{1,2,3}
Golfer 3	{2,3}	{1,2,3}	{1,2,3}	{1,2,3}
Golfer 4	{2,3}	{1,2,3}	{1,2,3}	{1,2,3}
Golfer 5	{2,3}	{1,2,3}	{1,2,3}	{1,2,3}
Golfer 6	{2,3}	{1,2,3}	{1,2,3}	{1,2,3}
Golfer 7	{2,3}	{1,2,3}	{1,2,3}	{1,2,3}
Golfer 8	{2,3}	{1,2,3}	{1,2,3}	{1,2,3}

Integer variables:

`assign[i,j]` variables whose value is from domain $\{1, 2, 3\}$

Constraints:

C1: each group has exactly groupSize players

C2: each pair of players only meets once

Constraint Programming: Representation

```
int: golfers = 9;
int: groupSize = 3;
int: days = 4;
int: groups = golfers/groupSize;
set of int: Golfer = 1..golfers;
set of int: Day = 1..days;
set of int: Group = 1..groups;

array[Golfer, Day] of var Group: assign; % Variables

constraint
  % C1: Each group has exactly groupSize players
  forall (gr in Group, d in Day)( % c1
   sum (g in Golfer) (bool2int(assign[g,d] = gr)) = groupSize
  ) /\
  % C2: Each pair of players only meets at most once
  forall (g1, g2 in Golfer, d1, d2 in Day  where g1 != g2 /\ d1 != d2) (
   (bool2int(assign[g1,d1] = assign[g2,d1]) + bool2int(assign[g2,d2] = assign[g2,d2])) <=1);

solve :: int_search([assign[i,j] | i in Golfer, j in Day ],
                    first_fail, indomain_min, complete)  satisfy;
```

Constraint Programming: Reasoning

The solution process proceeds by propagating the constraints on the domanins of the variables (ie, removing values) and tentatively assigning variables until only feasible values are left or backtracking.

Outline

20. Constraint Programming

21. Randomized Optimization Heuristics

Yet Another Example: TSP

Example (Traveling Salesman Problem)

Can you find a better solution?

Heuristics

- Get inspired by approach to problem solving in human mind
  [A. Newell and H.A. Simon. "Computer science as empirical inquiry: symbols and search." Communications of the ACM, ACM, 1976, 19(3)]
  - effective rules without theoretical support
  - trial and error
- Applications:
  - Optimization
  - But also in Psychology, Economics, Management [Tversky, A.; Kahneman, D. (1974). "Judgment under uncertainty: Heuristics and biases". Science 185]
- Basis on empirical evidence rather than mathematical logic. Getting things done in the given time.

Randomized Optimization Heuristics (ROHs)

Two main search paradigms:

- Constructive search
- Local search

plus high level guiding heuristics (ie, metaheuristics), eg, evolutionary algorithms.

ROHs: Representation

	Group 1	**Group 2**	**Group 3**
Day 0	0 1 2	3 4 5	6 7 8
Day 1	**0 4** 6	1 **3 7**	2 5 8
Day 2	**0 4** 8	1 5 6	2 **3 7**
Day 3	0 5 7	1 3 8	2 4 6

- Variables = solution representation, tentative solution
- Constraints: relaxed = soft
- Evaluation function to guide the search

ROHs: Reasoning, Local Search

Solution: Trial and Error

	Group 1	**Group 2**	**Group 3**
Day 0	0 1 2	3 4 5	6 7 8
Day 1	**0 4** 6	1 **3 7**	2 5 8
Day 2	**0 4** 8	1 5 6	2 **3 7**
Day 3	0 5 7	1 3 8	2 4 6

Heuristic algorithms: compute, efficiently, **good** solutions to a problem (without caring for theoretical guarantees on running time and approximation quality).

ROHs: Reasoning, Local Search

Example on Traveling Salesman Problem:

ROHs: Reasoning, Metaheuristics

Accepting worsening changes

Trying different changes

ROHs: Reasoning, Metaheuristics

- Stochastic Local Ssearch
- Simulated Annealing
- Iterated Local Search
- Tabu Search
- Variable Neighborhood Search
- Adaptive Large Neighborhood Search
- Evolutionary Algorithms
- Ant Colony Optimization
- Estimation-of-Distribution Algorithms
- Artificial Immune Systems
- ...
- Evolutionary Computation Bestiary `http://fcampelo.github.io/EC-Bestiary/`
- **Super**natural inspired [Maturana, Fouhey, 2013]

A Classification

- White box optimization:
  models can be expressed mathematically

- Grey box optimization:
  internal information about objective function computation is often available
  models that have a mathematical expression but may need data to determine them (eg, neural networks)

- Black box optimization:
  no mathematical expression is available

Approaches to ROHs

- White/Grey box: representation (modelling) + reasoning (search)
  constraint based local search, comet, local solver (Hexaly)

- Black Box: a different approach, framework separating problem from solvers and defining the interface specification
  EasyLocal, ...,

$\Longrightarrow$ Cost Action: ROAR-NET

ROAR-NET

https://github.com/roar-net/roar-net-api-spec

A Search Problem

Definition (Problem statement)

Assume we want to solve a **constrained optimization problem**: $\min f(x) \mid x \in F$ where is a set of **feasible solutions** and f an objective function. All parameters of the problem are known and deterministic.

Definition (Search or Optimization Algorithm)

Goal formulation: we want to find the minimum with respect to some criteria from a set of candidate elements.

Problem formulation: Given a description of the states, an initial state and actions necessary to reach the goal, find a sequence of actions to reach the goal.

Search: the algorithm simulates sequences of actions in the model of the goal, searching until it finds a sequence of actions that reaches the goal. The algorithm might have to simulate multiple tentative answers that do not meet the goal, but eventually it reach a solution, or it will find that no solution is possible.

Search Algorithms

Components of a Search Algorithm (1):

- **State or Search Space** A set of possible **states** that the search can be in.
- **State or (Candidate) solution**: a definition of the states of the search,
- **Initial State** that the search starts in. For example: an empty set of actions or a complete set of actions.
- **Goal** A set of one or more goal states. Sometimes there is one goal state sometimes there is a small set of alternative goal states
- **Evaluation function** $f(s)$ assess the distance from a potential goal. It can also include relaxed constraints.

Search Algorithms

Components of a Search Algorithm (2):

- **Action Type** t available to the algorithm. Neighborhood Structure
- For a given Action Type t and a State/Solution s, $Actions(t, s)$ returns a finite set of actions of type t that can be executed in s. We say that each of these actions is applicable in s. A transition model, which describes what each action does. Neighborhood
- **Result**(s, a) returns the state that results from doing action a in state s. Apply Move
- **Action-Cost**(s, a, s') **or** $c(s, a, s')$ action cost function gives the numeric cost of applying action a in state s to reach state s'. It reflects the evaluation of the state. Increment

Constraint Handling

In the Constraint Based Local Search community, **constraints** in heuristic methods are handled:

- implicitly in the definition of the search space and of the actions
- as one way constraints
- as soft constraints
  ie, relaxed in the evaluation function as objectives with large weights or as lexicographically more important objectives

Application Programming Interface (API)

(Here: not meant as Web API, network-based API, or REST API.)
The ROAR-NET API Specification is the definition of an interface or protocol between optimization problems seen as black box and their solvers in order to facilitate understanding, reusing and scaling of solution approaches.

We look for a model which

- ... allows one to use off the shelf components to solve it.
- ... assumes a separation between **problem specifics** and **solver**.
- .. is designed as a **software interface** offering a service to other pieces of software and is implemented by the user.
- ... promotes **reusability** of software components and minimizes the user's effort to deploy a solution for the specific optimization problem at hand.
- ... aims at maximizing code extensability, reusability, and simplicity.

Types

- **`Problem`** the problem instance
- **`Solution`** implements the representation of a **tentative** solution
- **`Value`** represents points in objective space
- **`Neighborhood`** a function that given a solution, gives another solution $neighbor : S \to S$, based on a neighborhood, compute a `Move`
- **`Move`** applied to a solution to get the novel solution
- **`Increment`** represents points in objective space

Simplification in single objective cases: **`Value`** and **`Increment`** are Reals (ie, **`Double`** or **`Integer`**)

Operations: Problem Representation

... on Problem (P):

- `empty_solution(P): Solution`
- `random_solution(P): Solution`
- `heuristic_solution(P): Solution[0..1]`

... on Solution S:

- `copy_solution(S): Solution`
- `lower_bound(S): Value[0..1]`
- `evaluation_value(): Value[0..1]`

Operations: Search

... on Neighborhood (N):

- `construction_neighbourhood(Problem): Neighbourhood`
- `destruction_neighbourhood(Problem): Neighbourhood`
- `local_neighbourhood(Problem): Neighbourhood`
- `moves(N, Solution): Move[0..*]`
- `random_move(N, Solution): Move[0..1]`
- `random_move_without_replacement(N, Solution): Move[0..*]`

Operations: Search

... on Move:

- `lower_bound_increment(Move, Solution): double[0..1]`
- `objective_value_increment(Move, Solution): double[0..1]`
  `apply_move(M, Solution): Solution` applies the move to a solution, to get a novel solution
- `invert_move(M): Move` computes the inverse of a move to revert it

The Full API

```
# Types
Problem
Solution
Value
Increment
PreferenceModel
Neighborhood
Move
```

```
# Operations on Problem
empty_solution
random_solution
heuristic_solution

# Operations on Solution
objective_value
lower_bound
copy_solution

# Operations on Neighborhood
local_neighbourhood
construction_neighbourhood
destruction_neighbourhood
```

```
# Operations on Move
moves
random_move
random_moves_without_replacement
lower_bound_increment
objective_value_increment
apply_move
invert_move

# Operations on PreferenceModel
scalarisation
better_or_indifferent  # preorder
better                 # strict preorder
indifferent            # equivalence
incomparable

compare_total_order
compare_partial_order

selection    # Indices
ranking      # Rank/class values
```

Single Machine Total Weighted Tardiness

Given: a set of n jobs $\{J_1, \ldots, J_n\}$ to be processed on a single machine and for each job J_i a processing time p_i, a weight w_i and a due date d_i.

Task: Find a schedule that minimizes the total weighted tardiness $\sum_{i=1}^{n} w_i \cdot T_i$ where $T_i = \max\{C_i - d_i, 0\}$ (C_i completion time of job J_i)

Example:

Job	J_1	J_2	J_3	J_4	J_5	J_6
Processing Time	3	2	2	3	4	3
Due date	6	13	4	9	7	17
Weight	2	3	1	5	1	2

Sequence $\phi = J_3, J_1, J_5, J_4, J_1, J_6$

Job	J_3	J_1	J_5	J_4	J_2	J_6
C_i	2	5	9	12	14	17
T_i	0	0	2	3	1	0
$w_i \cdot T_i$	0	0	2	15	3	0

Single Machine Total Weighted Tardiness Problem

- Interchange: size $\binom{n}{2}$ and $O(|i-j|)$ evaluation each
  - first-improvement: π_j, π_k

    $p_{\pi_j} \leq p_{\pi_k}$ for improvements, $w_j T_j + w_k T_k$ must decrease because jobs in $\pi_j, \ldots, \pi_k$ can only increase their tardiness.

    $p_{\pi_j} \geq p_{\pi_k}$ possible use of auxiliary data structure to speed up the computation
  - best-improvement: π_j, π_k

    $p_{\pi_j} \leq p_{\pi_k}$ for improvements, $w_j T_j + w_k T_k$ must decrease at least as the best interchange found so far because jobs in $\pi_j, \ldots, \pi_k$ can only increase their tardiness.

    $p_{\pi_j} \geq p_{\pi_k}$ possible use of auxiliary data structure to speed up the computation
- Swap: size $n-1$ and $O(1)$ evaluation each
- Insert: size $(n-1)^2$ and $O(|i-j|)$ evaluation each
  But possible to speed up with systematic examination by means of swaps: an interchange is equivalent to $|i-j|$ swaps hence overall examination takes $O(n^2)$

18. Metaheuristics

Remarks

See solutions to Sheet 11 on SMTWTP

- Logging
- Testing
- Multiple neighborhoods
- Metaheuristics

Outline

22. LS Based Metaheuristics
    - Randomized (or Stochastic) Local Search
    - Guided Local Search
    - Simulated Annealing
    - Iterated Local Search
    - Tabu Search
    - Variable Neighborhood Search
23. Construction Based Metaheuristics
    - Complete Search
    - Optimization Problems
    - Incomplete Search
    - Rollout/Pilot Method
    - Beam Search
    - GRASP
    - Iterated Greedy
24. Population Based Metaheuristics

Escaping Local Optima

Possibilities:

- **Restart:** re-initialize search whenever a local optimum is encountered.
  (Often rather ineffective due to cost of initialization.)
- **Non-improving steps**: in local optima, allow selection of candidate solutions with equal or worse evaluation function value, *e.g.*, using minimally worsening steps.
  (Can lead to long walks in **plateaus**, *i.e.*, regions of search positions with identical evaluation function.)
- **Diversify the neighborhood**

Note: None of these mechanisms is guaranteed to always escape effectively from local optima.

Diversification **vs** Intensification

- Goal-directed and randomized components of LS strategy need to be balanced carefully.
- **Intensification**: aims at greedily increasing solution quality, *e.g.*, by exploiting the evaluation function.
- **Diversification**: aims at preventing search stagnation, that is, the search process getting trapped in confined regions.

Examples:

- Iterative Improvement (II, First improvement or Best improvement): **intensification** strategy.
- Uninformed Random Walk/Picking (URW/P): **diversification** strategy.

Balanced combination of intensification and diversification mechanisms forms the basis for advanced LS methods.

Outline

22. LS Based Metaheuristics
    Randomized (or Stochastic) Local Search
    Guided Local Search
    Simulated Annealing
    Iterated Local Search
    Tabu Search
    Variable Neighborhood Search
23. Construction Based Metaheuristics
    Complete Search
    Optimization Problems
    Incomplete Search
    Rollout/Pilot Method
    Beam Search
    GRASP
    Iterated Greedy
24. Population Based Metaheuristics

Greedy Local Search

Key idea: Best improvement (Hill Climber or Steepest Descent) + Sideways Moves + seldom worsening moves

Algorithm 6.1: GSAT (F)

```
Input      : A CNF formula F
Parameters : Integers MAX-FLIPS, MAX-TRIES
Output     : A satisfying assignment for F, or FAIL
begin
    for i ← 1 to MAX-TRIES do
        σ ← a randomly generated truth assignment for F
        for j ← 1 to MAX-FLIPS do
            if σ satisfies F then return σ                    // success
            v ← a variable flipping which results in the greatest decrease
                (possibly negative) in the number of unsatisfied clauses
            Flip v in σ
    return FAIL                          // no satisfying assignment found
end
```

- GSAT begins with a rapid greedy descent towards a better truth assignment
- then long sequences of **sideways** moves take place. Sideways moves are moves that do not increase or decrease the total number of unsatisfied clauses. They navigate through **plateaux**, which is SAT are many and large
- GSAT [Selman et al. 1992] at its times was able to beat complete search algorithms (they were not using CDC)

Randomized Iterative Improvement

aka, Stochastic Hill Climbing

Key idea: Allow worsening moves: In each search step, with a fixed probability perform an uninformed random walk step instead of an iterative improvement step.
greedy + uniform random walk

Randomized Iterative Improvement (RII):
determine initial candidate solution s
while termination condition is not satisfied **do**
  With probability wp:
    choose a neighbor s' of s uniformly at random
  Otherwise:
    choose a neighbor s' of s such that $f(s') < f(s)$ or,
      if no such s' exists, choose s' such that $f(s')$ is minimal
  $s := s'$

With infite time it reaches optimum with $p > 0$ [Hoos and Tsang].

Example: Randomized Iterative Improvement for SAT

```
procedure RIISAT(F, wp, maxSteps)
   input: a formula F, probability wp, integer maxSteps
   output: a model φ for F or ∅
   choose assignment φ for F uniformly at random;
   steps := 0;
   while not(φ is not proper) and (steps < maxSteps) do
      with probability wp do
         select x in X uniformly at random and flip;
      otherwise
         select x in X^c uniformly at random from those that
            maximally decrease number of clauses violated;
      change φ;
      steps := steps+1;
   end
   if φ is a model for F then return φ
   else return ∅
   end
end RIISAT
```

X^c set of variables in violated clauses

Note:

- No need to terminate search when local minimum is encountered

  Instead: Impose limit on number of search steps or CPU time, from beginning of search or after last improvement.

- Probabilistic mechanism permits arbitrary long sequences of random walk steps

  Therefore: When run sufficiently long, RII is guaranteed to find (optimal) solution to any problem instance with arbitrarily high probability.

- GWSAT [Selman et al., 1994], was at some point state-of-the-art for SAT.

Focused Local Search: WalkSAT

```
procedure WalkSAT (F, maxTries, maxSteps, slc)
    input: CNF formula F, positive integers maxTries and maxSteps,
        heuristic function slc
    output: model of F or 'no solution found'
    for try := 1 to maxTries do
        a := randomly chosen assignment of the variables in formula F;
        for step := 1 to maxSteps do
            if a satisfies F then return a end
            c := randomly selected clause unsatisfied under a;
            x := variable selected from c according to heuristic function slc;
            a := a with x flipped;
        end
    end
    return 'no solution found'
end WalkSAT
```

Example of **slc** heuristic: with prob. wp select a random move, with prob. $1 - wp$ select the best

Extension to CSP. Recall the Definitions

Constraint Satisfaction Problem (CSP)

A CSP is a finite set of variables X, together with a finite set of constraints C, each on a subset of X. A **solution** to a CSP is an assignment of a value $d \in D(x)$ to each $x \in X$, such that all constraints are satisfied simultaneously.

Constraint Optimization Problem (COP)

A COP is a CSP P defined on the variables $x_1, \ldots, x_n$, together with an objective function $f : D(x_1) \times \cdots \times D(x_n) \to Q$ that assigns a value to each assignment of values to the variables. An **optimal solution** to a minimization (maximization) COP is a solution d to P that minimizes (maximizes) the value of $f(d)$.

$\rightsquigarrow$ Constraints in a CSP can be relaxed and their violations determine the objective function. This is the most common approach in LS

Min-Conflict Heuristic

```
procedure MCH (P, maxSteps)
    input: CSP instance P, positive integer maxSteps
    output: solution of P or "no solution found"

    a := randomly chosen assignment of the variables in P;
    for step := 1 to maxSteps do
        if a satisfies all constraints of P then return a end
        x := randomly selected variable from conflict set K(a);
        v := randomly selected value from the domain of x such that
             setting x to v minimises the number of unsatisfied constraints;
        a := a with x set to v;
    end
    return "no solution found"
end MCH
```

Outline

22. LS Based Metaheuristics
    - Randomized (or Stochastic) Local Search
    - Guided Local Search
    - Simulated Annealing
    - Iterated Local Search
    - Tabu Search
    - Variable Neighborhood Search
23. Construction Based Metaheuristics
    - Complete Search
    - Optimization Problems
    - Incomplete Search
    - Rollout/Pilot Method
    - Beam Search
    - GRASP
    - Iterated Greedy
24. Population Based Metaheuristics

Guided Local Search

- **Key Idea:** Modify the evaluation function whenever a local optimum is encountered.
- Associate **weights** (**penalties**) with solution components; these determine impact of components on evaluation function value.
- Perform Iterative Improvement; when in local minimum, increase penalties of some solution components until improving steps become available.

Guided Local Search (GLS):

determine **initial candidate solution** s
initialize penalties
while **termination criterion** is not satisfied **do**
  compute **modified evaluation function** g' from g
    based on **penalties**
  perform **subsidiary local search** on s
    using **evaluation function** g'
  update penalties based on s

Guided Local Search

- **Modified evaluation function:**

$$g'(s) := f(s) + \sum_{i \in SC(s)} \texttt{penalty}(i),$$

  where $SC(s)$ is the set of solution components used in candidate solution s.

- **Penalty initialization:** For all i: $\texttt{penalty}(i) := 0$.

- **Penalty update** in local minimum s: Typically involves **penalty increase** of some or all solution components of s; often also occasional **penalty decrease** or **penalty smoothing**.

- **Subsidiary local search:** Often **Iterative Improvement**.

Potential problem:
Solution components required for (optimal) solution may also be present in many local minima.

Possible solutions:

A: Occasional decreases/smoothing of penalties.

B: Only increase penalties of solution components that are least likely to occur in (optimal) solutions.

Implementation of **B**: Only increase penalties of solution components i with maximal utility [Voudouris and Tsang, 1995]:

$$\mathtt{util}(s, i) := \frac{f_i(s)}{1 + \mathtt{penalty}(i)}$$

where $f_i(s)$ is the solution quality contribution of i in s.

Example: Guided Local Search (GLS) for the TSP

[Voudouris and Tsang 1995; 1999]

- **Given:** TSP instance π
- **Search space:** Hamiltonian cycles in π with n vertices;
- **Neighborhood:** 2-edge-exchange;
- **Solution components** edges of π;
  $f_e(G, p) := w(e)$;
- **Penalty initialization:** Set all edge penalties to zero.
- **Subsidiary local search:** Iterative First Improvement.
- **Penalty update:** Increment penalties of all edges with maximal utility by

  $$\lambda := 0.3 \cdot \frac{w(s_{2\text{-}opt})}{n}$$

  where $s_{2\text{-}opt}$ = 2-optimal tour.

Guided Local Search for SAT

- Assign a positive weight to each clause
- attempt to minimize the sum of the weights of the unsatisfied clauses.
- The clause weights are dynamically modified (additively or multiplicatively) as the search progresses, increasing the weight of the clauses that are currently unsatisfied.
- Depends on:
  how often and by how much the weights of unsatisfied clauses are increased, and
  how are all weights periodically decreased in order to prevent certain weights from becoming dis-proportionately high.

Discrete Lagrangian Method

- Change the objective function bringing constraints g_i into it

$$L(\boldsymbol{s}, \boldsymbol{\lambda}) = f(\boldsymbol{s}) + \sum_i \lambda_i g_i(\boldsymbol{s})$$

- λ_i are continous variables called Lagrangian Multipliers
- $L(\boldsymbol{s}^*, \lambda) \leq L(\boldsymbol{s}^*, \boldsymbol{\lambda}^*) \leq L(\boldsymbol{s}, \boldsymbol{\lambda}^*)$
- Alternate optimizations in $\boldsymbol{s}$ and in $\boldsymbol{\lambda}$

Discrete Lagrangian Method for SAT

let $U_i(x)$ be a function that is 0 if C_i is satisfied by a solution x, and 1 otherwise.

$$\text{minimize } N(x) = \sum_{i=1}^{m} U_i(x)$$
$$\text{s.t.} U_i(x) = 0 \qquad \forall i \in \{1, 2, \ldots, m\}$$

Discrete Lagrangian Function:

$$L_d(x, \lambda) = N(x) + \sum_{i=1}^{m} \lambda_i U_i(x)$$

A point $(x^*, \lambda^*) \in \{0, 1\}^n \times \mathbb{R}^m$ is called a **saddle point** of the Lagrange function $L_d(x, \lambda)$ if it is a local minimum w.r.t. x^* and a local maximum w.r.t. λ^*. Formally, (x^*, λ^*) is a saddle point for L_d if $L_d(x^*, \lambda) \leq L_d(x^*, \lambda^*) \leq L_d(x, \lambda^*)$

Probabilistic Iterative Improv.

Key idea: Accept worsening steps with probability that depends on respective deterioration in evaluation function value: bigger deterioration $\cong$ smaller probability

Realization:

- Function $p(f, s)$: determines probability distribution over neighbors of s based on their values under evaluation function f.
- Accept s' neighbor of s with probability $p(f, s, s')$.

Note:

- Behavior of PII crucially depends on choice of p.
- II and RII are special cases of PII.

Example: Metropolis PII for the TSP

- **Search space** S**:** set of all Hamiltonian cycles in given graph G.
- **Solution set:** same as S
- **Neighborhood relation** $\mathcal{N}(s)$**:** 2-edge-exchange
- **Initialization:** an Hamiltonian cycle uniformly at random.
- **Step function:** implemented as 2-stage process:

  1. select neighbor $s' \in N(s)$ uniformly at random;
  2. accept as new search position with probability:

  $$p(T, s, s') := \begin{cases} 1 & \text{if } f(s') \leq f(s) \\ \exp \frac{-(f(s') - f(s))}{T} & \text{otherwise} \end{cases}$$

  (Metropolis condition), where **temperature** parameter T controls likelihood of accepting worsening steps.

- **Termination:** upon exceeding given bound on run-time.

Outline

22. LS Based Metaheuristics
    - Randomized (or Stochastic) Local Search
    - Guided Local Search
    - Simulated Annealing
    - Iterated Local Search
    - Tabu Search
    - Variable Neighborhood Search
23. Construction Based Metaheuristics
    - Complete Search
    - Optimization Problems
    - Incomplete Search
    - Rollout/Pilot Method
    - Beam Search
    - GRASP
    - Iterated Greedy
24. Population Based Metaheuristics

Inspired by statistical mechanics in matter physics:

- candidate solutions $\cong$ states of physical system
- evaluation function $\cong$ thermodynamic energy
- globally optimal solutions $\cong$ ground states
- parameter $T \cong$ physical temperature

Note: In physical process (*e.g.*, annealing of metals), perfect ground states are achieved by very slow lowering of temperature.

Simulated Annealing

Key idea: Vary temperature parameter, *i.e.*, probability of accepting worsening moves, in Probabilistic Iterative Improvement according to annealing schedule (aka **cooling schedule**).

Simulated Annealing (SA):

determine initial candidate solution s
set initial temperature T according to annealing schedule
while termination condition is not satisfied: **do**
  while maintain same temperature T according to annealing schedule **do**
    probabilistically choose a neighbor s' of s using proposal mechanism
    if s' satisfies probabilistic acceptance criterion (depending on T) **then**
      $s := s'$
  update T according to annealing schedule

- 2-stage step function based on
  - proposal mechanism (often uniform random choice from $N(s)$)
  - acceptance criterion (often **Metropolis condition**)

- Annealing schedule
  (function mapping run-time t onto temperature $T(t)$):
  - initial temperature T_0
    (may depend on properties of given problem instance)
  - temperature update scheme
    (*e.g.*, linear cooling: $T_{i+1} = T_0(1 - i/I_{max})$,
    geometric cooling: $T_{i+1} = \alpha \cdot T_i$)
  - number of search steps to be performed at each temperature
    (often multiple of neighborhood size)
  - may be **static** or **dynamic**
  - seek to balance moderate execution time with asymptotic behavior properties

- Termination predicate: often based on **acceptance ratio**,
  i.e., ratio accepted / proposed steps **or** number of idle iterations

Example: Simulated Annealing for TSP

Extension of previous PII algorithm for the TSP, with

- proposal mechanism: uniform random choice from 2-exchange neighborhood;
- acceptance criterion: Metropolis condition (always accept improving steps, accept worsening steps with probability $\exp[-(f(s') - f(s))/T]$);
- annealing schedule: geometric cooling $T := 0.95 \cdot T$ with $n \cdot (n-1)$ steps at each temperature (n = number of vertices in given graph), T_0 chosen such that 97% of proposed steps are accepted;
- termination: when for five successive temperature values no improvement in solution quality and acceptance ratio $< 2\%$.

Improvements:

- neighborhood pruning (*e.g.*, candidate lists for TSP)
- greedy initialization (*e.g.*, by using NNH for the TSP)
- **low temperature starts** (to prevent good initial candidate solutions from being too easily destroyed by worsening steps)

Profiling

Run A
Run B
Temperature
Cost function value
Iterations/10^7
Iterations/10^7

Outline

22. LS Based Metaheuristics
    - Randomized (or Stochastic) Local Search
    - Guided Local Search
    - Simulated Annealing
    - Iterated Local Search
    - Tabu Search
    - Variable Neighborhood Search
23. Construction Based Metaheuristics
    - Complete Search
    - Optimization Problems
    - Incomplete Search
    - Rollout/Pilot Method
    - Beam Search
    - GRASP
    - Iterated Greedy
24. Population Based Metaheuristics

Iterated Local Search

Key Idea: Use two types of LS steps:

- **subsidiary local search** steps for reaching local optima as efficiently as possible (intensification)
- **perturbation steps** for effectively escaping from local optima (diversification).

Also: Use **acceptance criterion** to control diversification **vs** intensification behavior.

Iterated Local Search (ILS):
determine initial candidate solution s
perform **subsidiary local search** on s
while termination criterion is not satisfied **do**
  $r := s$
  perform **perturbation** on s
  perform **subsidiary local search** on s
  based on **acceptance criterion**,
    keep s or revert to $s := r$

Note:

- **Subsidiary local search** results in a local minimum.
- ILS trajectories can be seen as walks in the space of local minima of the given evaluation function.
- **Perturbation phase** and **acceptance criterion** may use aspects of **search history** (*i.e.*, limited memory).
- In a high-performance ILS algorithm, **subsidiary local search**, **perturbation mechanism** and **acceptance criterion** need to complement each other well.

Components

Subsidiary local search:

- More effective subsidiary local search procedures lead to better ILS performance.
  Example: 2-opt **vs** 3-opt **vs** LK for TSP.

- Often, subsidiary local search = iterative improvement,
  but more sophisticated LS methods can be used.
  (*e.g.*, Tabu Search).

Components

Perturbation mechanism:

- Needs to be chosen such that its effect **cannot** be easily undone by subsequent local search phase.
  (Often achieved by search steps larger neighborhood.)
  Example: local search = 3-opt, perturbation = 4-exchange steps in ILS for TSP.
- A perturbation phase may consist of one or more perturbation steps.
- Weak perturbation $\Rightarrow$ short subsequent local search phase;
  but: risk of revisiting current local minimum.
- Strong perturbation $\Rightarrow$ more effective escape from local minima;
  but: may have similar drawbacks as random restart.
- Advanced ILS algorithms may change nature and/or strength of perturbation adaptively during search.

Components

Acceptance criteria:

- Always accept the **best** of the two candidate solutions

  $\Rightarrow$ ILS performs Iterative Improvement in the space of local optima reached by subsidiary local search.

- Always accept the **most recent** of the two candidate solutions

  $\Rightarrow$ ILS performs random walk in the space of local optima reached by subsidiary local search.

- Intermediate behavior: select between the two candidate solutions based on the **Metropolis criterion** (*e.g.*, used in **Large Step Markov Chains** [Martin et al., 1991].

- Advanced acceptance criteria take into account search history,
  e.g., by occasionally reverting to **incumbent solution**.

Examples

Example: Iterated Local Search for the TSP (1)

- **Given:** TSP instance π.
- **Search space:** Hamiltonian cycles in π.
- **Subsidiary local search:** Lin-Kernighan variable depth search algorithm
- **Perturbation mechanism:**
  'double-bridge move' = particular 4-exchange step:

- **Acceptance criterion:** Always return the best of the two given candidate round trips.

Outline

22. LS Based Metaheuristics
    - Randomized (or Stochastic) Local Search
    - Guided Local Search
    - Simulated Annealing
    - Iterated Local Search
    - Tabu Search
    - Variable Neighborhood Search
23. Construction Based Metaheuristics
    - Complete Search
    - Optimization Problems
    - Incomplete Search
    - Rollout/Pilot Method
    - Beam Search
    - GRASP
    - Iterated Greedy
24. Population Based Metaheuristics

Tabu Search

Key idea: Avoid repeating history (memory)
How can we remember the history without

- memorizing full solutions (space)
- computing hash functions (time)

$\rightsquigarrow$ use attributes

Tabu Search

Key idea: Use aspects of search history (memory) to escape from local minima.

- Associate **tabu attributes** with candidate solutions or solution components.
- Forbid steps to search positions recently visited by underlying iterative best improvement procedure based on tabu attributes.

Tabu Search (TS):

determine initial candidate solution s
While **termination criterion** is not satisfied:
  determine set N' of non-tabu neighbors of s
  choose a best candidate solution s' in N'

  update tabu attributes based on s'
  $s := s'$

Example: Tabu Search for CSP

- **Search space:** set of all complete assignments of X.
- **Neighborhood structure:** one exchange
- **Memory:** Associate tabu status (Boolean value) with each pair (variable,value) (x, val).
- **Initialization:** a random assignment
- **Search steps:**
  - pairs (x, v) are tabu if they have been changed in the last `tt` steps;
  - neighboring assignments are admissible if they can be reached by changing a non-tabu pair or have fewer unsatisfied constraints than the best assignments seen so far (aspiration criterion);
  - choose uniformly at random admissible neighbors with minimal number of unsatisfied constraints.
- **Termination:** upon finding a feasible assignment **or**
  after given bound on number of search steps has been reached **or**
  after a number of idle iterations

Note:

- **Admissible neighbors of** s: Non-tabu search positions in $N(s)$
- **Tabu tenure**: a fixed number of subsequent search steps
  for which the last search position
  or the solution components just added/removed from it
  are declared **tabu**
- **Aspiration criterion** (often used): specifies conditions under which
  tabu status may be overridden (*e.g.*, if considered step leads to improvement in incumbent solution).
- Crucial for efficient implementation:
  - efficient **best improvement** local search
    $\rightsquigarrow$ pruning, delta updates, (auxiliary) data structures
  - efficient determination of tabu status:
    store for each variable x the number of the search step
    when its value was last changed it_x; x is tabu if
    $it - it_x < tt$, where it = current search step number.

Design Choices

Design choices:

- Neighborhood exploration:
  - no reduction
  - min-conflict heuristic
- Prohibition power for `move = <x,new_v,old_v>`
  - `<x,-,->`
  - `<x,-,old_v>`
  - `<x,new_v,old_v>`, `<x,old_v,new_v>`
- Tabu list dynamics:
  - Interval: $\mathtt{tt} \in [t_b, t_b + w]$
  - Adaptive: $\mathtt{tt} = \lfloor \alpha \cdot c \rfloor + \mathtt{RandU}(0, t_b)$

Outline

22. LS Based Metaheuristics
    - Randomized (or Stochastic) Local Search
    - Guided Local Search
    - Simulated Annealing
    - Iterated Local Search
    - Tabu Search
    - Variable Neighborhood Search
23. Construction Based Metaheuristics
    - Complete Search
    - Optimization Problems
    - Incomplete Search
    - Rollout/Pilot Method
    - Beam Search
    - GRASP
    - Iterated Greedy
24. Population Based Metaheuristics

Variable Neighborhood Search

Variable Neighborhood Search is a method based on the systematic change of the neighborhood during the search.

Central observations

- a local minimum w.r.t. one neighborhood function is not necessarily locally minimal w.r.t. another neighborhood function
- a global optimum is locally optimal w.r.t. **all** neighborhood functions

Key principle: change the neighborhood during the search

- Several adaptations of this central principle
  - (Basic) Variable Neighborhood Descent (VND)
  - Variable Neighborhood Search (VNS)
  - Reduced Variable Neighborhood Search (RVNS)
  - Variable Neighborhood Decomposition Search (VNDS)
  - Skewed Variable Neighborhood Search (SVNS)
- Notation
  - N_k, $k = 1, 2, \ldots, k_m$ is a set of neighborhood functions
  - $N_k(s)$ is the set of solutions in the k-th neighborhood of s

How to generate the various neighborhood functions?

- for many problems different neighborhood functions (local searches) exist / are in use
- change parameters of existing local search algorithms
- use k-exchange neighborhoods; these can be naturally extended
- many neighborhood functions are associated with distance measures; in this case increase the distance

Basic Variable Neighborhood Descent

Procedure BVND
input : N_k, $k = 1, 2, \ldots, k_{max}$, and an initial solution s
output: a local optimum s for N_k, $k = 1, 2, \ldots, k_{max}$
$k \leftarrow 1$
repeat
  $s' \leftarrow$ FindBestNeighbor(s,N_k)
  if $f(s') < f(s)$ **then**
    $s \leftarrow s'$
    $(k \leftarrow 1)$
  else
    $k \leftarrow k + 1$
until $k = k_{max}$;

Variable Neighborhood Descent

Procedure VND
input : N_k, $k = 1, 2, \ldots, k_{max}$, and an initial solution s
output: a local optimum s for N_k, $k = 1, 2, \ldots, k_{max}$
$k \leftarrow 1$
repeat
  $s' \leftarrow$ **IterativeImprovement**(s, N_k)
  if $f(s') < f(s)$ **then**
    $s \leftarrow s'$
    $k \leftarrow 1$
  else
    $k \leftarrow k + 1$
until $k = k_{max}$;

- Final solution is locally optimal w.r.t. all neighborhoods
- First improvement may be applied instead of best improvement
- Typically, order neighborhoods from smallest to largest
- If iterative improvement algorithms II_k, $k = 1, \ldots, k_{max}$ are available as black-box procedures:
  - order black-boxes
  - apply them in the given order
  - possibly iterate starting from the first one
  - order chosen by: **solution quality** and **speed**

Basic Variable Neighborhood Search

Procedure BVNS
input : N_k, $k = 1, 2, \ldots, k_{max}$, and an initial solution s
output: a local optimum s for N_k, $k = 1, 2, \ldots, k_{max}$

```
repeat
    k ← 1
    repeat
        s' ← RandomPicking(s, N_k)
        s'' ← IterativeImprovement(s', N_k)
        if f(s'') < f(s) then
            s ← s''
            k ← 1
        else
            k ← k + 1
    until k = k_max;
until Termination Condition;
```

To decide:

- which neighborhoods
- how many
- which order
- which change strategy

- Extended version: parameters k_{min} and k_{step}; set $k \leftarrow k_{min}$ and increase by k_{step} if no better solution is found (achieves diversification)

Outline

22. LS Based Metaheuristics
    - Randomized (or Stochastic) Local Search
    - Guided Local Search
    - Simulated Annealing
    - Iterated Local Search
    - Tabu Search
    - Variable Neighborhood Search
23. Construction Based Metaheuristics
    - Complete Search
    - Optimization Problems
    - Incomplete Search
    - Rollout/Pilot Method
    - Beam Search
    - GRASP
    - Iterated Greedy
24. Population Based Metaheuristics

Constructive search

What is a **partial** solution (as opposed to a **complete** solution)?

- Solutions as subsets of a larger **ground set of solution components**
- Partial solutions as a representation of all candidate solutions that contain them
- Not all subsets of components are valid partial solutions
- Construction rule
- Assessment of partial solutions:
  - inferred from the sets of solutions that they represent
  - Lower bound (minimization) or upper bound (maximization)

Outline

22. LS Based Metaheuristics
    Randomized (or Stochastic) Local Search
    Guided Local Search
    Simulated Annealing
    Iterated Local Search
    Tabu Search
    Variable Neighborhood Search

23. Construction Based Metaheuristics
    Complete Search
    Optimization Problems
    Incomplete Search
    Rollout/Pilot Method
    Beam Search
    GRASP
    Iterated Greedy

24. Population Based Metaheuristics

Complete Search Methods

Tree (or graph) search in

Uninformed settings (satisfaction probs)

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
- Bidirectional Search

Informed settings (optimization probs)

- best-first search, aka, greedy search
- A^* search
- Iterative Deepening A^*
- Memory bounded A^*
- Recursive best first

In construction heuristics for this course, we can assume tree search of fixed known depth.

Complete Tree Search

Uninformed

Search Space

tree with branching factor at the top level nd
at the next level $(n-1)d$.
The tree has $n! \cdot d^n$ even if only d^n possible complete assignments.

Informed

- CSP is **commutative** in the order of application of any given set of action. (we reach same partial solution regardless of the order)
- Hence generate successors by considering possible assignments for only a single variable at each node in the search tree.
- look-ahead, best first, etc.

Dealing with Constraints

Backtracking search

depth-first search that chooses one variable at a time and backtracks when a variable has no legal values left to assign.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
     if value is consistent with assignment according to CONSTRAINTS[csp] then
        add {var = value} to assignment
        result ← RECURSIVE-BACKTRACKING(assignment, csp)
        if result ≠ failure then return result
        remove {var = value} from assignment
  return failure
```

Backtrack Search

- No need to copy solutions all the times but rather extensions and undo extensions
- Since CSP is standard then the alg is also standard and can use general purpose algorithms for initial state, successor function and goal test.
- Backtracking is **uninformed and complete**. Other search algorithms may use **information** in form of heuristics.

Backtracking
General Concepts

Decisions in general purpose methods:

1) Which variable should we assign next, and in what order should its values be tried?

2) What are the implications of the current variable assignments for the other unassigned variables?

3) When a path fails – that is, a state is reached in which a variable has no legal values can the search avoid repeating this failure in subsequent paths?

Search (1) + Inference (2) + Backtracking (3) = Constraint Programming

In the general case, at point 1) we use heuristic rules.

If we do not backtrack (point 3) then we have a **construction heuristic**.

1) **Which variable should we assign next, and in what order should its values be tried?**

- Select-Initial-Unassigned-Variable
- Select-Unassigned-Variable
  - most constrained first = fail-first heuristic
    = Minimum remaining values (MRV) heuristic
    (tend to reduce the branching factor and to speed up pruning)
  - least constrained last

  Eg.: max degree, farthest, earliest due date, etc.
- Order-Domain-Values
  - greedy
  - least constraining value heuristic
    (leaves maximum flexibility for subsequent variable assignments)
  - maximal regret
    implements a kind of look ahead

2) **What are the implications of the current variable assignments for the other unassigned variables?**

Propagating information through constraints:

- Implicit in Select-Unassigned-Variable
- Forward checking (coupled with Minimum Remaining Values)
- Constraint propagation in CSP
  - arc consistency: force all (directed) arcs uv to be consistent:
    $\exists$ a value in $D(v)$: $\forall$ values in $D(u)$, otherwise detects inconsistency

    can be applied as preprocessing or as propagation step after each assignment (Maintaining Arc Consistency)

    Applied repeatedly

3) **When a path fails – that is, a state is reached in which a variable has no legal values can the search avoid repeating this failure in subsequent paths?**

Backtracking-Search

- chronological backtracking, the most recent decision point is revisited
- backjumping, backtracks to the most recent variable in the conflict set (set of previously assigned variables connected to X by constraints).

Outline

22. LS Based Metaheuristics
    - Randomized (or Stochastic) Local Search
    - Guided Local Search
    - Simulated Annealing
    - Iterated Local Search
    - Tabu Search
    - Variable Neighborhood Search
23. Construction Based Metaheuristics
    - Complete Search
    - Optimization Problems
    - Incomplete Search
    - Rollout/Pilot Method
    - Beam Search
    - GRASP
    - Iterated Greedy
24. Population Based Metaheuristics

Dealing with Objectives

A* search

- The priority assigned to a node x is determined by the function

$$f(x) = g(x) + h(x)$$

  $g(x)$: cost of the path so far
  $h(x)$: heuristic estimate of the minimal cost to reach the goal from x.

- It is optimal if $h(x)$ is an
  - admissible heuristic: **never overestimates** the cost to reach the goal
  - consistent: $h(n) \leq c(n, a, n') + h(n')$
    (consistent $\implies$ admissible, only necessary in graph search)

A* best-first search

Figure 3.2 A simplified road map of part of Romania.

Figure 4.3 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 4.1.

A* search

Possible choices for admissible heuristic functions

- optimal solution to an easily solvable **relaxed problem**
- optimal solution to an easily solvable **subproblem**
- learning from experience by gathering statistics on state features
- preferred heuristics functions with higher values (provided they do not overestimate)
- if several heuristics available $h_1, h_2, \ldots, h_m$ and not clear which is the best then:

$$h(x) = \max\{h_1(x), \ldots, h_m(x)\}$$

A* search

Drawbacks

- Time complexity: In the worst case, the number of nodes expanded is exponential, (but it is polynomial when the heuristic function h meets the following condition:

$$|h(x) - h^*(x)| \leq O(\log h^*(x))$$

  h^* is the optimal heuristic, the exact cost of getting from x to the goal.)

- Memory usage: In the worst case, it must remember an exponential number of nodes. Several variants: including iterative deepening A* (IDA*), memory-bounded A* (MA*) and simplified memory bounded A* (SMA*) and recursive best-first search (RBFS)

Incomplete Search

Complete search is often better suited when ...

- proofs of insolubility or optimality are required;
- time constraints are not critical;
- problem-specific knowledge can be exploited.

Incomplete search is the necessary choice when ...

- non linear constraints and non linear objective function;
- reasonably good solutions are required within a short time;
- problem-specific knowledge is rather limited.

Outline

22. LS Based Metaheuristics
    - Randomized (or Stochastic) Local Search
    - Guided Local Search
    - Simulated Annealing
    - Iterated Local Search
    - Tabu Search
    - Variable Neighborhood Search
23. Construction Based Metaheuristics
    - Complete Search
    - Optimization Problems
    - Incomplete Search
    - Rollout/Pilot Method
    - Beam Search
    - GRASP
    - Iterated Greedy
24. Population Based Metaheuristics

Greedy algorithms

Greedy algorithms

- Strategy: always make the choice that is best at the moment
- They are not generally guaranteed to find globally optimal solutions
  (but sometimes they do: Minimum Spanning Tree, Single Source Shortest Path, etc.)

We will see problem sepcific examples

Best-first search

Figure 3.2 A simplified road map of part of Romania.

Figure 4.2 Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

Incomplete Search

On backtracking framework
(beyond depth-first search)

- Bounded backtrack
- Credit-based search
- Limited Discrepancy Search
- Barrier Search
- Randomization in Tree Search
- Random Restart

Outside the exact framework
(beyond greedy search)

- Random Restart
- Rollout/Pilot Method
- Beam Search
- Iterated Greedy
- GRASP
- (Adaptive Iterated Construction Search)
- (Multilevel Refinement)

Bounded backtrack

Bounded-backtrack search:

bbs(10)

Depth-bounded, then bounded-backtrack search:

dbs(2, bbs(0))

http://4c.ucc.ie/~hsimonis/visualization/techniques/partial_search/main.htm

Credit-based search

- Key idea: important decisions are at the top of the tree
- **Credit** = backtracking steps
- Credit distribution: one half at the best child the other divided among the other children.
- When credits run out follow deterministic best-search
- In addition: allow limited backtracking steps (eg, 5) at the bottom
- **Control parameters**: initial credit, distribution of credit among the children, amount of local backtracking at bottom.

Limited Discrepancy Search

Limited Discrepancy Search (LDS)

- Key observation that often the heuristic used in the search is nearly always correct with just a few exceptions.
- Explore the tree in increasing number of **discrepancies**, modifications from the heuristic choice.
- Eg: count one discrepancy if second best is chosen
  count two discrepancies either if third best is chosen or twice the second best is chosen
- **Control parameter**: the number of discrepancies

Randomization in Tree Search

The idea comes from complete search: the important decisions are made up in the search tree (backdoors - set of variables such that once they are instantiated the remaining problem simplifies to a tractable form)
$\rightsquigarrow$ random selections + restart strategy

Random selections

- randomization in variable ordering:
  - breaking ties at random
  - use heuristic to rank and randomly pick from small factor from the best
  - random pick among heuristics
  - random pick variable with probability depending on heuristic value
- randomization in value ordering:
  - just select random from the domain

Restart strategy in backtracking

- Example: $S_u = (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 4, 8, 1, \ldots)$

Outline

22. LS Based Metaheuristics
    - Randomized (or Stochastic) Local Search
    - Guided Local Search
    - Simulated Annealing
    - Iterated Local Search
    - Tabu Search
    - Variable Neighborhood Search
23. Construction Based Metaheuristics
    - Complete Search
    - Optimization Problems
    - Incomplete Search
    - Rollout/Pilot Method
    - Beam Search
    - GRASP
    - Iterated Greedy
24. Population Based Metaheuristics

Rollout/Pilot Method

Derived from A*

- Each candidate solution is a collection of m components $S = (s_1, s_2, \ldots, s_m)$.
- Master process adds components sequentially to a partial solution $S_k = (s_1, s_2, \ldots s_k)$
- At the k-th iteration the master process evaluates feasible components to add based on an **heuristic look-ahead strategy**.
- The evaluation function $H(S_{k+1})$ is determined by sub-heuristics that complete the solution starting from S_k
- Sub-heuristics are combined in $H(S_{k+1})$ by
  - weighted sum
  - minimal value

Note: this evaluation is not a lower bound!

Speed-ups:

- halt whenever cost of current partial solution exceeds current upper bound
- evaluate only a fraction of possible components

Outline

22. LS Based Metaheuristics
    - Randomized (or Stochastic) Local Search
    - Guided Local Search
    - Simulated Annealing
    - Iterated Local Search
    - Tabu Search
    - Variable Neighborhood Search
23. Construction Based Metaheuristics
    - Complete Search
    - Optimization Problems
    - Incomplete Search
    - Rollout/Pilot Method
    - Beam Search
    - GRASP
    - Iterated Greedy
24. Population Based Metaheuristics

Beam Search

Based on the tree search framework:

- maintain a set B of bw (beam width) partial candidate solutions
- at each iteration extend each solution from B in fw (filter width) possible ways
- rank each $bw \times fw$ candidate solutions and take the best bw partial solutions
- complete candidate solutions obtained by B are maintained in B_f
- Stop when no partial solution in B is to be extended

Example

Three phases of Beam Search (expansion, simulation, pruning), illustrated with beam width $bw = 3$ and expansion factor $bf = 2$.

In the 'Expansion' a tree consisting of bw blue trajectories represents a beam extended to depth 2. Of all its possible child nodes, only bf candidates (colored red) per each leaf node are selected. Wavy arrows in the 'Simulation' figure complete the rollouts, each earning a different reward value. In the 'Pruning' phase, the beam grows by another depth, keeping its width size to bf after pruning out the candidate nodes with poor performances.

Choo, Jinho et al. 2022

Outline

22. LS Based Metaheuristics
    - Randomized (or Stochastic) Local Search
    - Guided Local Search
    - Simulated Annealing
    - Iterated Local Search
    - Tabu Search
    - Variable Neighborhood Search
23. Construction Based Metaheuristics
    - Complete Search
    - Optimization Problems
    - Incomplete Search
    - Rollout/Pilot Method
    - Beam Search
    - GRASP
    - Iterated Greedy
24. Population Based Metaheuristics

GRASP

Greedy Randomized Adaptive Search Procedure

Key Idea: Combine randomized constructive search with subsequent local search.

Motivation:

- Candidate solutions obtained from construction heuristics can often be substantially improved by local search.
- Local search methods often require substantially fewer steps to reach high-quality solutions when initialized using greedy constructive search rather than random picking.
- By iterating cycles of constructive + local search, further performance improvements can be achieved.

Greedy Randomized "Adaptive" Search Procedure (GRASP):
while **termination criterion** is not satisfied **do**
  generate candidate solution s using
    subsidiary greedy randomized constructive search
  perform **subsidiary local search** on s

- Randomization in **constructive search** ensures that a large number of good starting points for **subsidiary local search** is obtained.
- Constructive search in GRASP is 'adaptive' (or dynamic): Heuristic value of solution component to be added to a given partial candidate solution may depend on solution components present in it.
- Variants of GRASP without local search phase (aka **semi-greedy heuristics**) typically do not reach the performance of GRASP with local search.

Restricted candidate lists (RCLs)

- Each step of **constructive search** adds a solution component selected uniformly at random from a **restricted candidate list (RCL)**.
- RCLs are constructed in each step using a **heuristic function** h.
  - RCLs based on **cardinality restriction** comprises the k best-ranked solution components. (k is a parameter of the algorithm.)
  - RCLs based on **value restriction** comprise all solution components l for which $h(l) \leq h_{min} + \alpha \cdot (h_{max} - h_{min})$, where h_{min} = minimal value of h and h_{max} = maximal value of h for any l. (α is a parameter of the algorithm.)
  - Possible extension: **reactive GRASP** (*e.g.*, dynamic adaptation of α during search)

Example: Squeaky Wheel

Key idea: solutions can reveal problem structure which maybe worth to exploit.

Use a greedy heuristic repeatedly by prioritizing the elements that create troubles.

Squeaky Wheel

- **Constructor**: greedy algorithm on a sequence of problem elements.
- **Analyzer**: assign a penalty to problem elements that contribute to flaws in the current solution.
- **Prioritizer**: uses the penalties to modify the previous sequence of problem elements. Elements with high penalty are moved toward the front.

Possible to include a local search phase between one iteration and the other

Outline

22. LS Based Metaheuristics
    - Randomized (or Stochastic) Local Search
    - Guided Local Search
    - Simulated Annealing
    - Iterated Local Search
    - Tabu Search
    - Variable Neighborhood Search
23. Construction Based Metaheuristics
    - Complete Search
    - Optimization Problems
    - Incomplete Search
    - Rollout/Pilot Method
    - Beam Search
    - GRASP
    - Iterated Greedy
24. Population Based Metaheuristics

Iterated Greedy

Key idea: use greedy construction

- alternation of **construction** and **deconstruction** phases
- an acceptance criterion decides whether the search continues from the new or from the old solution.

Iterated Greedy (IG):

determine initial candidate solution s
while termination criterion is not satisfied **do**
  $r := s$
  (randomly or heuristically) **destruct** part of s
  greedily **reconstruct** the missing part of s
  based on **acceptance criterion**,
    keep s or revert to $s := r$

Adaptive Large Neighborhood Search

https://imada.sdu.dk/u/marco/DM841/slides/dm841-hr-alns.pdf

Outline

22. LS Based Metaheuristics
    - Randomized (or Stochastic) Local Search
    - Guided Local Search
    - Simulated Annealing
    - Iterated Local Search
    - Tabu Search
    - Variable Neighborhood Search
23. Construction Based Metaheuristics
    - Complete Search
    - Optimization Problems
    - Incomplete Search
    - Rollout/Pilot Method
    - Beam Search
    - GRASP
    - Iterated Greedy
24. Population Based Metaheuristics

Ant Colony Optimization

https://imada.sdu.dk/u/marco/DM841/slides/dm841-hr-aco.pdf

Evolutionary Algorithms

https://imada.sdu.dk/u/marco/DM841/slides/dm841-hr-evolutionary.pdf