Differentiable Optimisation in Deep Learning

Stephen Gould
stephen.gould@anu.edu.au

Australian National University

February 2026

Today Optimization is Everywhere

- **financial mathematics:** maximize profits or minimize costs subject to constraints on resources and budgets

Today Optimization is Everywhere

- **financial mathematics:** maximize profits or minimize costs subject to constraints on resources and budgets
- **mechanical engineering:** maximize the span of a suspension bridge subject to load constraints; maximize the lift-to-drag ratio of an aeroplane wing

Today Optimization is Everywhere

- **financial mathematics:** maximize profits or minimize costs subject to constraints on resources and budgets
- **mechanical engineering:** maximize the span of a suspension bridge subject to load constraints; maximize the lift-to-drag ratio of an aeroplane wing
- **electrical engineering:** minimize the size of a transistor in a circuit subject to power and timing constraints

Today Optimization is Everywhere

- **financial mathematics:** maximize profits or minimize costs subject to constraints on resources and budgets
- **mechanical engineering:** maximize the span of a suspension bridge subject to load constraints; maximize the lift-to-drag ratio of an aeroplane wing
- **electrical engineering:** minimize the size of a transistor in a circuit subject to power and timing constraints
- **logistics and planning:** find the cheapest way to distribute goods from suppliers to consumers across a network

Today Optimization is Everywhere

- **financial mathematics:** maximize profits or minimize costs subject to constraints on resources and budgets
- **mechanical engineering:** maximize the span of a suspension bridge subject to load constraints; maximize the lift-to-drag ratio of an aeroplane wing
- **electrical engineering:** minimize the size of a transistor in a circuit subject to power and timing constraints
- **logistics and planning:** find the cheapest way to distribute goods from suppliers to consumers across a network
- **statistics/data science:** curve fitting and data visualization

Today Optimization is Everywhere

- **financial mathematics:** maximize profits or minimize costs subject to constraints on resources and budgets
- **mechanical engineering:** maximize the span of a suspension bridge subject to load constraints; maximize the lift-to-drag ratio of an aeroplane wing
- **electrical engineering:** minimize the size of a transistor in a circuit subject to power and timing constraints
- **logistics and planning:** find the cheapest way to distribute goods from suppliers to consumers across a network
- **statistics/data science:** curve fitting and data visualization
- **robotics:** optimise control parameters to achieve some goal state or trajectory; simultaneous localisation and mapping (SLAM); point/feature matching

Today Optimization is Everywhere

- **financial mathematics:** maximize profits or minimize costs subject to constraints on resources and budgets
- **mechanical engineering:** maximize the span of a suspension bridge subject to load constraints; maximize the lift-to-drag ratio of an aeroplane wing
- **electrical engineering:** minimize the size of a transistor in a circuit subject to power and timing constraints
- **logistics and planning:** find the cheapest way to distribute goods from suppliers to consumers across a network
- **statistics/data science:** curve fitting and data visualization
- **robotics:** optimise control parameters to achieve some goal state or trajectory; simultaneous localisation and mapping (SLAM); point/feature matching
- **machine learning and deep learning:** minimize loss functions with respect to the parameters of our model

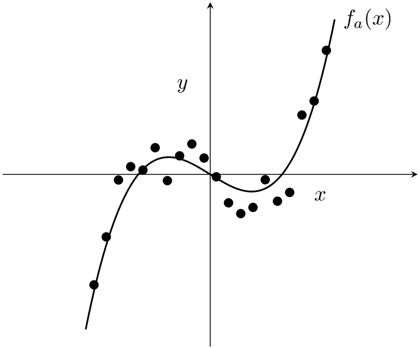
Classic Example: Polynomial Curve Fitting

fit n-th order polynomial $f_a(x) = \sum_{k=0}^{n} a_k x^k$ to set of noisy points $\{(x_i, y_i)\}_{i=1}^{m}$
(here a_k are the variables, and x_i and y_i are the data)

Classic Example: Polynomial Curve Fitting

fit n-th order polynomial $f_a(x) = \sum_{k=0}^{n} a_k x^k$ to set of noisy points $\{(x_i, y_i)\}_{i=1}^{m}$
(here a_k are the variables, and x_i and y_i are the data)

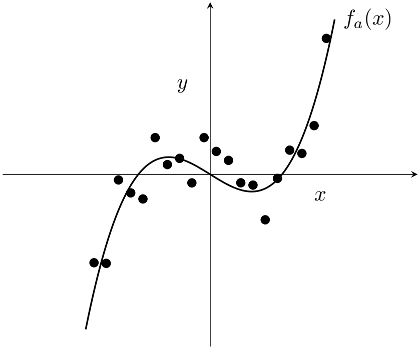
minimize (over a) $\quad \sum_{i=1}^{m} \left(f_a(x_i) - y_i\right)^2$



Classic Example: Polynomial Curve Fitting

fit n-th order polynomial $f_a(x) = \sum_{k=0}^{n} a_k x^k$ to set of noisy points $\{(x_i, y_i)\}_{i=1}^{m}$
(here a_k are the variables, and x_i and y_i are the data)

minimize (over a) $\quad \sum_{i=1}^{m} \left(f_a(x_i) - y_i\right)^2$



- later we will see that this is an instance of a least-squares problem, itself a type of convex optimisation problem

Overview

Introduction to Optimisation (Part 1)

- Formal definition
- Least squares
- Convex sets and functions
- Convex optimisation problems
- Lagrangian
- Duality
- Optimality conditions
- Algorithms

Differentiable Optimisation and Deep Learning (Part 2)

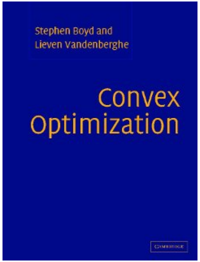
- Machine learning from 10,000ft
- Automatic differentiation
- Forward and backward passes
- Imperative and declarative nodes
- Bi-level optimisation
- Implicit function theorem
- Differentiable optimisation results
- Examples and applications

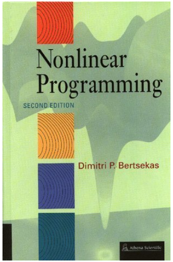
accompanying lecture notes available at

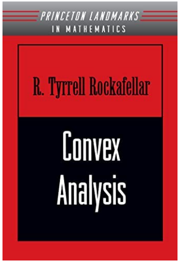
`https://users.cecs.anu.edu.au/~sgould`

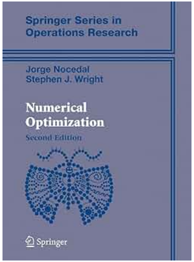
part 1

Part 1: Introduction to Optimisation

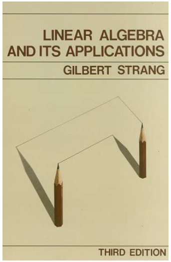


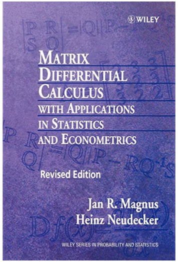


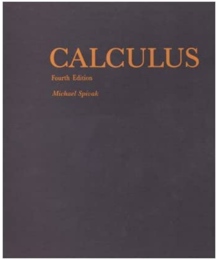




Assumed Background







Optimisation Problems

find an assignment to variables that minimises a measure of cost subject to some constraints[1]

[1] In these lectures we will be concerned with continuous-valued variables

Optimisation Problems

$$\begin{array}{ll} \text{minimize (over } x) & \text{objective}(x) \\ \text{subject to} & \text{constraints}(x) \end{array}$$

Optimisation Problems

$$
\begin{array}{ll}
\text{minimize} & f_0(x) \\
\text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, p \\
& h_i(x) = 0, \quad i = 1, \ldots, q
\end{array}
$$

- $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ — optimisation variables
- $f_0 : \mathbb{R}^n \to \mathbb{R}$ — objective (or cost or loss) function
- $f_i : \mathbb{R}^n \to \mathbb{R}$, $i = 1, \ldots, p$ — inequality constraint functions
- $h_i : \mathbb{R}^n \to \mathbb{R}$, $i = 1, \ldots, q$ — equality constraint functions

Solution and Optimal Value

A point x is **feasible** if $x \in \mathbf{dom}\,(f_0)$ and it satisfies the constraints.

A **solution**, or optimal point, $x^\star$ has the smallest value of f_0 among all feasible x.

[1]Warning: notation clash between p and $p^\star$!

Solution and Optimal Value

A point x is **feasible** if $x \in \mathbf{dom}\,(f_0)$ and it satisfies the constraints.

A **solution**, or optimal point, $x^\star$ has the smallest value of f_0 among all feasible x.

The **optimal value** is[1]

$$p^\star = \inf_{x \in \mathcal{D}} \left\{ f_0(x) \;\middle|\; \begin{array}{ll} f_i(x) \leq 0, & i = 1, \ldots, p \\ h_i(x) = 0, & i = 1, \ldots, q \end{array} \right\}.$$

- $p^\star$ and is equal to $f_0(x^\star)$ when $x^\star$ exists
- $p^\star = \infty$ if the problem is infeasible (no x satisfies the constraints)
- $p^\star = -\infty$ if the problem is unbounded below

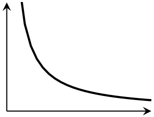
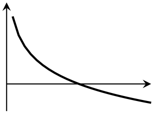
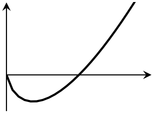
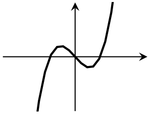
[1] Warning: notation clash between p and $p^\star$!

Locally Optimal Points

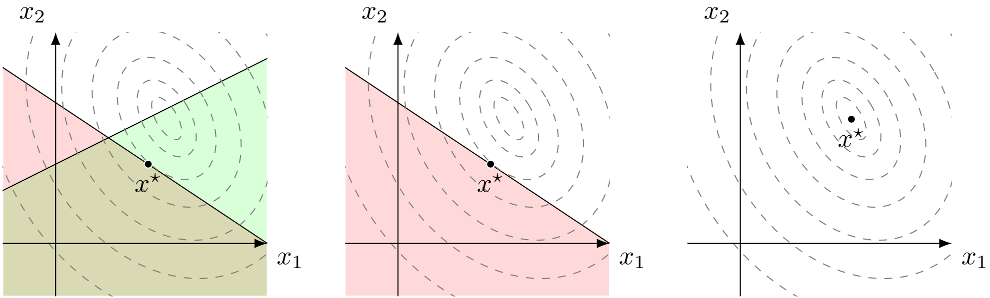
A point x is **locally optimal** if there is an $R > 0$ such that $z = x$ is optimal for

$$\begin{array}{lll} \text{minimize (over } z) & f_0(z) & \\ \text{subject to} & f_i(z) \leq 0 & i = 1, \ldots, p \\ & h_i(z) = 0 & i = 1, \ldots, q \\ & \|z - x\|_2 \leq R. & \end{array}$$

Examples (1D)

	$1/x$	$-\log x$	$x \log x$	$x^3 - 3x$
f_0:				
dom (f_0):	$\mathbb{R}_{++}$	$\mathbb{R}_{++}$	$\mathbb{R}_{++}$	$\mathbb{R}$
$p^\star$:	0	$-\infty$	$-1/e$	$-\infty$
$x^\star$:	none	none	$1/e$	$x = 1$ locally

Examples (2D)



Least Squares

$$\text{minimize} \quad \|Ax - b\|_2^2$$

Least Squares

$$\text{minimize} \quad \|Ax - b\|_2^2$$

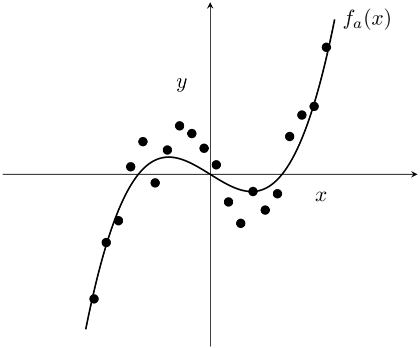
- unique solution if A^TA is invertible, $x^\star = \left(A^TA\right)^{-1}A^Tb$
- solution via SVD, $A = U\Sigma V^T$, if A^TA not invertible, $x^\star = V\Sigma^{-1}U^Tb$
  - in fact, $x^\star + w$ for any $w \in \mathcal{N}(A)$ also a solution
- solution via QR factorisation, $x^\star = R^{-1}Q^Tb$
- solved in $O(n^2m)$ time, less if structured
- typically use iterative solver (for large scale problems)

Reminder: Polynomial Curve Fitting Example

fit n-th order polynomial $f_a(x) = \sum_{k=0}^{n} a_k x^k$ to set of noisy points $\{(x_i, y_i)\}_{i=1}^{m}$

minimize (over a) $\quad \sum_{i=1}^{m} \left(f_a(x_i) - y_i\right)^2$

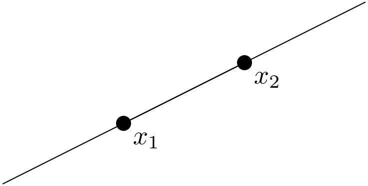
$$\text{minimize} \quad \left\| \underbrace{\begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^n \\ 1 & x_2 & x_2^2 & \dots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \dots & x_m^n \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}}_{x} - \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}}_{b} \right\|_2^2$$



Lines and Line Segments

- a **line** through two points x_1 and x_2

$$x = \theta x_1 + (1-\theta)x_2, \quad (\theta \in \mathbb{R})$$

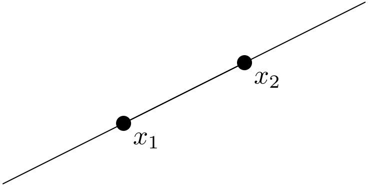


- an **affine set** contains the line through any two distinct points in the set
- an **affine hull** the set formed by taking all lines through points in a set

Lines and Line Segments

- a **line** through two points x_1 and x_2

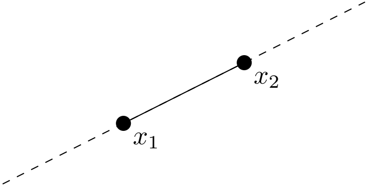
$$x = \theta x_1 + (1-\theta)x_2, \quad (\theta \in \mathbb{R})$$



- an **affine set** contains the line through any two distinct points in the set
- an **affine hull** the set formed by taking all lines through points in a set

- a **line segment** between x_1 and x_2

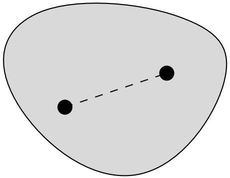
$$x = \theta x_1 + (1-\theta)x_2, \quad (0 \le \theta \le 1)$$

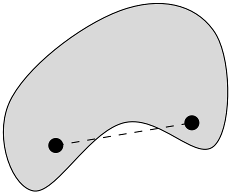


- a **convex set** contains the line segment between any two distinct points in the set
- an **convex hull** the set formed by taking all line segments between points in a set

Convex Sets

$$x_1, x_2 \in \text{convex set } C \quad \Longrightarrow \quad \theta x_1 + (1-\theta)x_2 \in C \text{ for all } 0 \le \theta \le 1$$

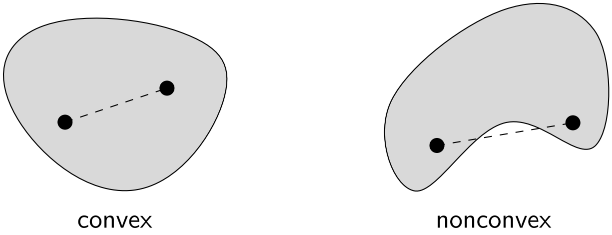




every point in C can "see" every other point in C

Convex Sets

$$x_1, x_2 \in \text{convex set } C \quad \Longrightarrow \quad \theta x_1 + (1-\theta)x_2 \in C \text{ for all } 0 \leq \theta \leq 1$$

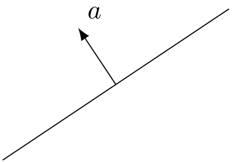


every point in C can "see" every other point in C

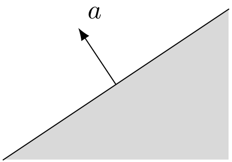
common examples in machine learning:

- nonnegative orthant, $\mathbb{R}^n_+ = \{x \mid x_i \geq 0, i = 1, \ldots, n\}$
- positive semindefinite matrices, $\mathbb{S}^n_+ = \{X \mid z^T X z \geq 0, z \in \mathbb{R}^n\}$

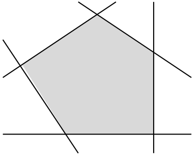
More Examples



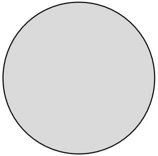
hyperplane,
$\{x \mid a^T x = b\}$



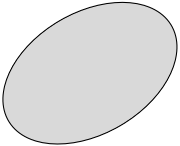
halfspace,
$\{x \mid a^T x \le b\}$



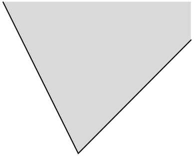
polyhedron,
$\{x \mid Ax \preceq b, Cx = d\}$



norm ball,
$\{x \mid \|x - x_c\|_p \le r\}$



ellipsoid,
$\{Au + b \mid \|u|_2 \le 1\}$



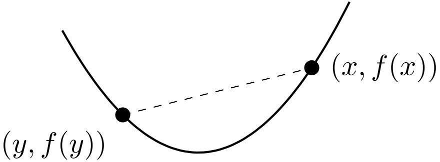
Lorentz cone,
$\{(x, t) \mid \|x\| \le t\}$

Convex Functions

A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if **dom** (f) is a convex set and

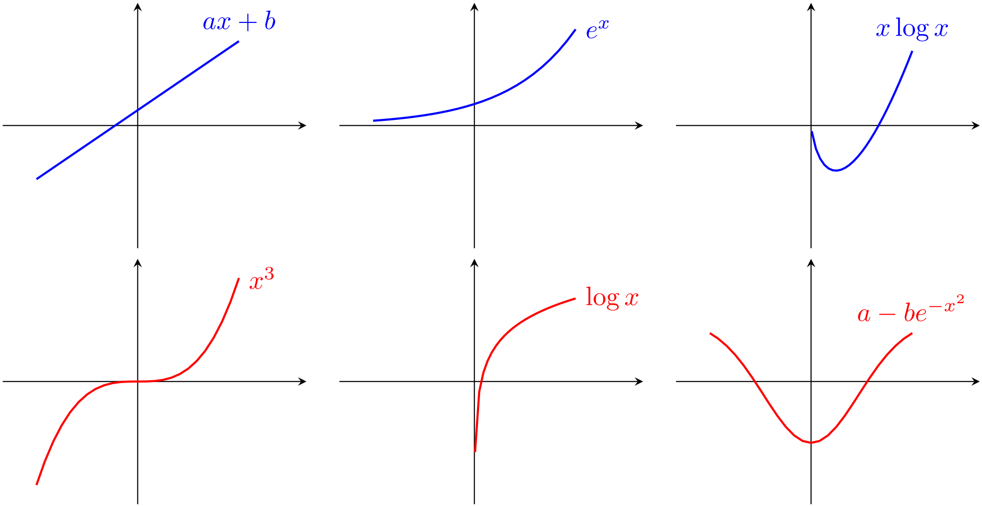
$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta) f(y)$$

for all $x, y \in$ **dom** (f) $, 0 \leq \theta \leq 1$.

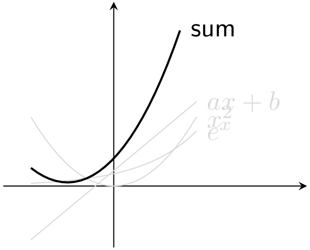


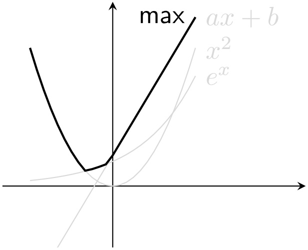
- f is concave if $-f$ is convex

Examples

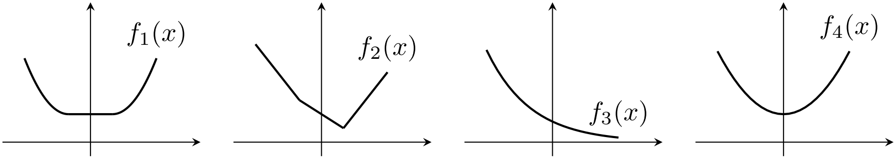


Weighted Sum and Pointwise Maximum Preserve Convexity





Convex, Strictly Convex, and Strongly Convex

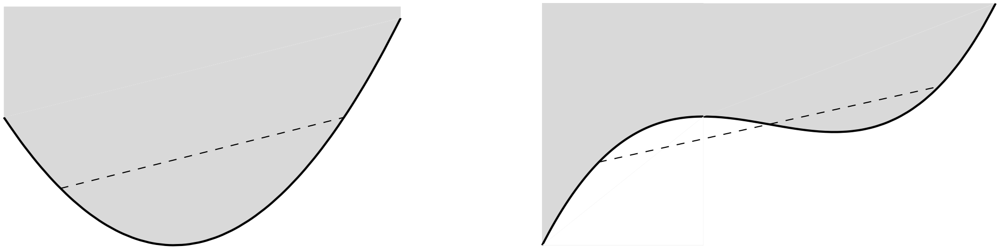


- f_1 is smooth and convex: $f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)y$
- f_2 is non-differentiable and convex: $f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)y$
- f_3 is strictly convex: $f(\theta x + (1-\theta)y) < \theta f(x) + (1-\theta)y$
- f_4 is strongly convex: $\exists m$ s.t. $m(y-x)^2 \leq f(y) - f(x)$

Epigraph

The epigraph of function $f : \mathbb{R}^n \to \mathbb{R}$ is the set

$$\textbf{epi}(f) = \{(x, t) \in \mathbb{R}^{n+1} \mid x \in \textbf{dom}\,(f)\,, f(x) \le t\}.$$

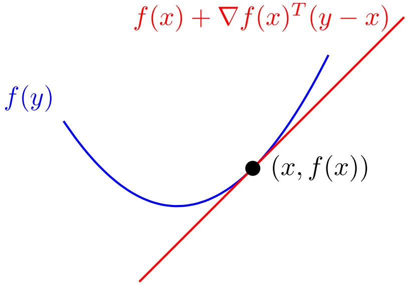


- f is a convex function if and only if $\textbf{epi}(f)$ is a convex set

First-order Condition

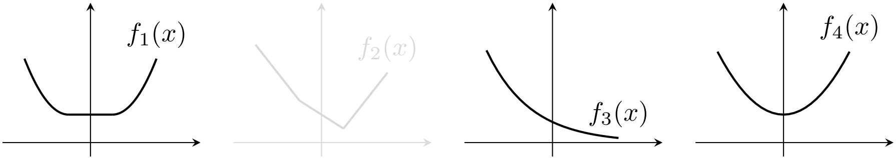
differentiable f with convex domain is convex iff

$$f(y) \geq f(x) + \nabla f(x)^T (y - x) \quad \text{for all } x, y \in \textbf{dom}\,(f)$$



- first-order approximation of (convex) f is a global under estimator

Second-order Condition



twice differentiable f with convex domain is convex iff

$$\nabla^2 f(x) \succeq 0 \quad \text{for all } x \in \textbf{dom}\,(f)$$

- if $\nabla^2 f(x) \succ 0$ for all $x \in \textbf{dom}\,(f)$, then f is strictly convex
- if $\nabla^2 f(x) \succeq mI$ for some $m > 0$ and all $x \in \textbf{dom}\,(f)$, then f is strongly convex
- strongly convex functions have a unique minimum

Example: log-sum-exp

The second-order condition can be used to establish the convexity of

$$f(x) = \log \sum_{k=1}^{n} \exp x_k$$

Example: log-sum-exp

The second-order condition can be used to establish the convexity of

$$f(x) = \log \sum_{k=1}^{n} \exp x_k$$

Proof Sketch.

- Compute Hessian (matrix of partial second derivatives),

$$\nabla^2 f(x) = \frac{1}{\left(\mathbf{1}^T z\right)^2} \left((\mathbf{1}^T z)\mathbf{diag}(z) - zz^T\right) \qquad (z_k = \exp x_k)$$

- Use the Cauchy-Schwarz inequality to show

$$v^T \left((\mathbf{1}^T z)\mathbf{diag}(z) - zz^T\right) v \geq 0, \quad \text{for all } v \in \mathbb{R}^n$$

implying that $\nabla^2 f(x) \succeq 0$

Convex Optimisation Problems

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, p \\ & a_i^T x = b_i, \quad i = 1, \ldots, q \end{array}$$

- $f_0, f_1, \ldots, f_p$ are convex
- $h_i(x) \triangleq a_i^T x - b_i$ are affine, often written as $Ax = b$
- the **feasible set** $\mathcal{X}$ is the set of all points x in the domain of $f_0, f_1, \ldots, f_p$ and that satisfy the constraints

minimise a convex objective over a convex feasible set

Convex Optimisation Problem Families

Linear Program (LP)

$$
\begin{array}{ll}
\text{minimize} & c^T x + d \\
\text{subject to} & Gx \preceq h \\
& Ax = b
\end{array}
$$

Convex Optimisation Problem Families

Linear Program (LP)

$$\begin{array}{ll} \text{minimize} & c^T x + d \\ \text{subject to} & Gx \preceq h \\ & Ax = b \end{array}$$

Quadratic Program (QP)

$$\begin{array}{ll} \text{minimize} & \frac{1}{2} x^T P x + q^T x + r \\ \text{subject to} & Gx \preceq h \\ & Ax = b \end{array}$$

where $P \succeq 0$.

Convex Optimisation Problem Families

Linear Program (LP)

$$\begin{array}{ll} \text{minimize} & c^T x + d \\ \text{subject to} & Gx \preceq h \\ & Ax = b \end{array}$$

Second-order Cone Program (SOCP)

$$\begin{array}{lll} \text{minimize} & f^T x & \\ \text{subject to} & \|A_i x + b_i\|_2 \leq c_i^T x + d_i, & i = 1, \ldots, m \\ & Fx = g & \end{array}$$

Quadratic Program (QP)

$$\begin{array}{ll} \text{minimize} & \frac{1}{2} x^T P x + q^T x + r \\ \text{subject to} & Gx \preceq h \\ & Ax = b \end{array}$$

where $P \succeq 0$.

Convex Optimisation Problem Families

Linear Program (LP)

$$\begin{array}{ll} \text{minimize} & c^T x + d \\ \text{subject to} & Gx \preceq h \\ & Ax = b \end{array}$$

Quadratic Program (QP)

$$\begin{array}{ll} \text{minimize} & \frac{1}{2} x^T P x + q^T x + r \\ \text{subject to} & Gx \preceq h \\ & Ax = b \end{array}$$

where $P \succeq 0$.

Second-order Cone Program (SOCP)

$$\begin{array}{lll} \text{minimize} & f^T x & \\ \text{subject to} & \|A_i x + b_i\|_2 \leq c_i^T x + d_i, & i = 1, \ldots, m \\ & Fx = g & \end{array}$$

Semidefinite Program (SDP)

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & x_1 F_1 + \cdots + x_n F_n + G \preceq 0 \\ & Ax = b \end{array}$$

where $G, F_1, \ldots, F_n \in \mathbb{S}^k$.

Local and Global Optima

any local minimum of a convex problem is (globally) optimal

Local and Global Optima

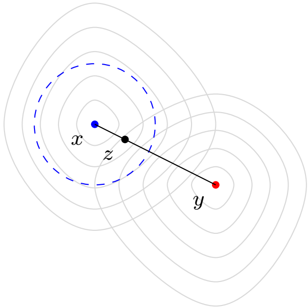
any local minimum of a convex problem is (globally) optimal

Proof Sketch.

- towards contradiction, suppose x is locally optimal, but there exists a feasible y with lower objective
- since x is locally optimally there exists a radius R such that no other point within R of x has lower objective
- (so y must be further than R from x)
- pick a point z on the line segment between x and y and within R of x
- so z must be feasible and have objective no lower than x
- but, by the basic inequality of convex functions,

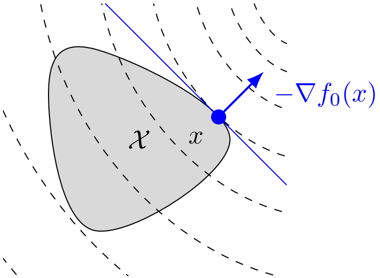
$$f_0(\theta x + (1-\theta)y) \leq \theta f_0(x) + (1-\theta) f_0(y),$$

  the objective value at z must be between that at x and y, i.e., lower than $f_0(x)$ (since $f_0(y) < f_0(x)$)
- we have a contradiction



Optimality Criterion for Differentiable f_0

x is optimal if and only if it is feasible and $\nabla f_0(x)^T(y-x) \geq 0$ for all feasible y



if nonzero,

- $\nabla f_0(x)$ defines a supporting hyperplane to feasible set $\mathcal{X}$ at x
- f_0 cannot be improved by moving in a direction where x stays feasible

Lagrangian

Standard form problem (not necessarily convex),

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, p \\ & h_i(x) = 0, \quad i = 1, \ldots, q \end{array}$$

variable $x \in \mathbb{R}^n$, domain $\mathcal{D}$, optimal value $p^\star$

Lagrangian

Standard form problem (not necessarily convex),

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \le 0, \quad i = 1, \dots, p \\ & h_i(x) = 0, \quad i = 1, \dots, q \end{array}$$

variable $x \in \mathbb{R}^n$, domain $\mathcal{D}$, optimal value $p^\star$

Lagrangian: $\mathcal{L} : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^q \to \mathbb{R}$, with $\textbf{dom}\,(\mathcal{L}) = \mathcal{D} \times \mathbb{R}^p \times \mathbb{R}^q$,

$$\mathcal{L}(x, \lambda, \nu) = f_0(x) + \sum_{i=1}^{p} \lambda_i f_i(x) + \sum_{i=1}^{q} \nu_i h_i(x)$$

- weighted sum of objective and constraint functions
- λ_i is the Lagrange multiplier (dual variable) associated with $f_i(x) \le 0$
- ν_i is the Lagrange multiplier (dual variable) associated with $h_i(x) = 0$

Lagrange Dual Function

Define Lagrange dual function, $g : \mathbb{R}^p \times \mathbb{R}^q \to \mathbb{R}$, as

$$\begin{aligned} g(\lambda, \nu) &= \inf_{x \in \mathcal{D}} \mathcal{L}(x, \lambda, \nu) \\ &= \inf_{x \in \mathcal{D}} \left(f_0(x) + \sum_{i=1}^{p} \lambda_i f_i(x) + \sum_{i=1}^{q} \nu_i h_i(x) \right) \end{aligned}$$

- g is concave (always), can be $-\infty$ for some λ, ν
- **lower bound property:** if $\lambda \succeq 0$, then $g(\lambda, \nu) \leq p^\star$ (since for feasible x we have $f_i(x) \leq 0$ and $h_i(x) = 0$)

The Dual Problem

The Lagrange dual problem is to maximise the dual function

$$\begin{array}{ll} \text{maximize} & g(\lambda, \nu) \\ \text{subject to} & \lambda \succeq 0 \end{array}$$

- finds the best lower bound on $p^\star$, obtained from Lagrange dual function
- a convex optimisation problem with optimal value denoted by $d^\star$
- λ, ν are dual feasible if $\lambda \succeq 0$ and $(\lambda, \nu) \in \textbf{dom}\,(g)$
- original problem is known as the **primal problem**

Weak and Strong Duality

weak duality: $d^\star \leq p^\star$

- always holds (for convex and nonconvex problems)
- can be used to find nontrivial lower bounds for difficult problems

strong duality: $d^\star = p^\star$

- does not hold in general
- (usually) holds for convex problems (e.g., all LPs and QPs)
- conditions that guarantee strong duality on convex problems are called **constraint qualifications**, e.g., Slater's condition

Karush-Kuhn-Tucker (KKT) Conditions

The following four conditions are called KKT conditions (for differentiable f_i, h_i):

- primal feasible: $\begin{array}{ll} f_i(x) \leq 0, & i = 1, \ldots, p \\ h_i(x) = 0, & i = 1, \ldots, q \end{array}$
- dual feasible: $\lambda \succeq 0$
- complementary slackness: $\lambda_i f_i(x) = 0$ for $i = 1, \ldots, p$
- gradient of Lagrangian with respect to x vanishes,

$$\nabla f_0(x) + \sum_{i=1}^{p} \lambda_i \nabla f_i(x) + \sum_{i=1}^{q} \nu_i \nabla h_i(x) = 0$$

Generalizes optimality condition $\nabla f_0(x) = 0$ for unconstrained problems.

Gradient Descent

$$\text{minimize} \quad f_0(x)$$

- f_0 convex, twice continuously differentiable
- we assume optimal value $p^\star = \inf_x f_0(x)$ is attained (and finite)

Gradient Descent

$$\text{minimize} \quad f_0(x)$$

- f_0 convex, twice continuously differentiable
- we assume optimal value $p^\star = \inf_x f_0(x)$ is attained (and finite)

Gradient descent:

1. **given** a starting point $x \in \textbf{dom}\,(f_0)$
2. **repeat** $x := x - t\nabla f_0(x)$. (choose step size, t)
3. **until** stopping criterion satisfied, e.g., $\|\nabla f_0(x)\|_2 \leq \epsilon$.

- variants of gradient descent define step direction Δx different to $-\nabla f_0(x)$

Choosing Step Size

fixed schedule: set t to a small constant or decay with each iteration

exact line search: $t = \text{argmin}_{t>0} f_0(x + t\Delta x)$

backtracking line search (with parameters $\alpha \in (0, 1/2), \beta \in (0, 1)$)

- starting at $t = 1$ with search direction Δx, repeat $t := \beta t$ until

$$f_0(x + t\Delta x) < f_0(x) + \alpha t \nabla f_0(x)^T \Delta x$$

Choosing Step Size

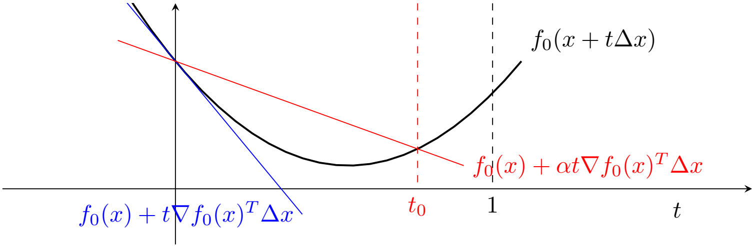
fixed schedule: set t to a small constant or decay with each iteration

exact line search: $t = \text{argmin}_{t>0} f_0(x + t\Delta x)$

backtracking line search (with parameters $\alpha \in (0, 1/2), \beta \in (0, 1)$)

- starting at $t = 1$ with search direction Δx, repeat $t := \beta t$ until

$$f_0(x + t\Delta x) < f_0(x) + \alpha t \nabla f_0(x)^T \Delta x$$



Choosing Step Size

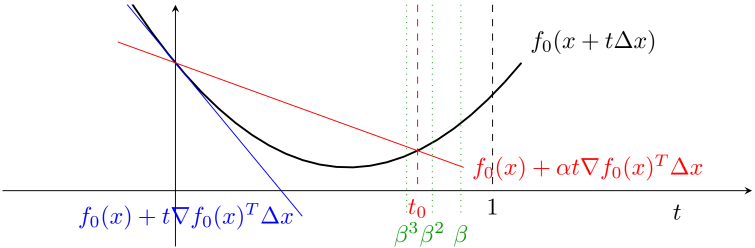
fixed schedule: set t to a small constant or decay with each iteration

exact line search: $t = \text{argmin}_{t>0} f_0(x + t\Delta x)$

backtracking line search (with parameters $\alpha \in (0, 1/2), \beta \in (0, 1)$)

- starting at $t = 1$ with search direction Δx, repeat $t := \beta t$ until

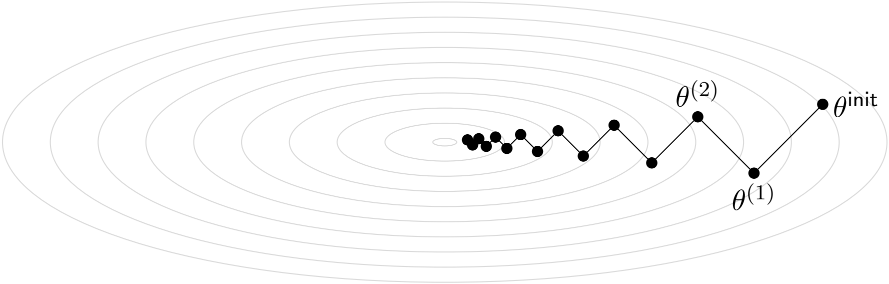
$$f_0(x + t\Delta x) < f_0(x) + \alpha t \nabla f_0(x)^T \Delta x$$



Example

Gradient descent (even with exact line search) can be slow. E.g.,

$$f_0(x) = x_1^2 + \gamma x_2^2, \quad \gamma \gg 1$$



Newton's Method

$$\Delta x_{\mathsf{nt}} = -\nabla^2 f_0(x)^{-1} \nabla f_0(x)$$

- $x + \Delta x_{\mathsf{nt}}$ minimizes the second-order approximation of f_0 at x,

$$\hat{f}(x+v) = f_0(x) + \nabla f_0(x)^T v + \frac{1}{2} v^T \nabla^2 f_0(x) v$$

Newton's method:

1. **given** a starting point $x \in \textbf{dom}\,(f_0)$.
2. **repeat** $x := x + t\Delta x_{\mathsf{nt}}$. (choose step size, t)
3. **until** stopping criterion satisfied.

Equality Constrained Methods

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & Ax = b \end{array}$$

- f_0 convex, twice continuously differentiable
- $A \in \mathbb{R}^{q \times n}$ with $\textbf{rank}(A) = q$ (and $b \in \textbf{range}(A)$)
- we assume $p^\star$ is finite and attained

Equality Constrained Methods

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & Ax = b \end{array}$$

- f_0 convex, twice continuously differentiable
- $A \in \mathbb{R}^{q \times n}$ with $\mathbf{rank}(A) = q$ (and $b \in \mathbf{range}(A)$)
- we assume $p^\star$ is finite and attained

optimality condition: $x^\star$ is optimal iff there exists a $\nu^\star$ such that

$$\nabla f_0(x^\star) + A^T \nu^\star = 0, \quad Ax^\star = b$$

Newton Step for Equality Constrained Optimisation

Newton step Δx_{nt} of f_0 at feasible x is given by solution v of

$$\begin{bmatrix} \nabla^2 f_0(x) & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} v \\ w \end{bmatrix} = \begin{bmatrix} -\nabla f_0(x) \\ 0 \end{bmatrix}$$

- second row ensures that x iterates stay feasible
- solves quadratic approximation of optimisation problem

$$\begin{array}{ll} \text{minimize} & \hat{f}(x+v) \triangleq f_0(x) + \nabla f_0(x)^T v + \frac{1}{2} v^T \nabla^2 f_0(x) v \\ \text{subject to} & A(x+v) = b \end{array}$$

- solves linear approximation of optimality condition

The Barrier Method

For inequality constrained problems,

$$
\begin{array}{ll}
\text{minimize} & f_0(x) \\
\text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, p \\
& Ax = b
\end{array}
$$

The Barrier Method

For inequality constrained problems,

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, p \\ & Ax = b \end{array}$$

we reformulate using an indicator function,

$$\begin{array}{ll} \text{minimize} & f_0(x) + \sum_{i=1}^{p} I_{\mathbb{R}_-}(f_i(x)) \\ \text{subject to} & Ax = b \end{array}$$

where $I_{\mathbb{R}_-}(u) = 0$ if $u \leq 0$ and $I_{\mathbb{R}_-}(u) = \infty$ otherwise,

The Barrier Method

For inequality constrained problems,

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, p \\ & Ax = b \end{array}$$

we reformulate using an indicator function,

$$\begin{array}{ll} \text{minimize} & f_0(x) + \sum_{i=1}^{p} I_{\mathbb{R}_-}(f_i(x)) \\ \text{subject to} & Ax = b \end{array}$$

where $I_{\mathbb{R}_-}(u) = 0$ if $u \leq 0$ and $I_{\mathbb{R}_-}(u) = \infty$ otherwise,
which we approximate with a logarithmic barrier

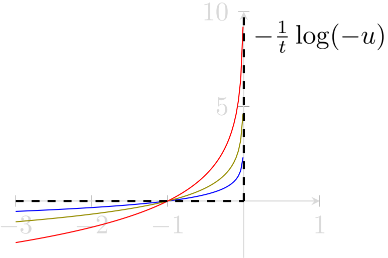
$$\begin{array}{ll} \text{minimize} & f_0(x) - \frac{1}{t} \sum_{i=1}^{p} \log(-f_i(x)) \\ \text{subject to} & Ax = b \end{array}$$

to get an equality constrained approximation.

The Barrier Method

For inequality constrained problems,

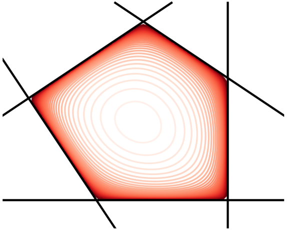
$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \dots, p \\ & Ax = b \end{array}$$



we reformulate using an indicator function,

$$\begin{array}{ll} \text{minimize} & f_0(x) + \sum_{i=1}^{p} I_{\mathbb{R}_-}(f_i(x)) \\ \text{subject to} & Ax = b \end{array}$$

where $I_{\mathbb{R}_-}(u) = 0$ if $u \leq 0$ and $I_{\mathbb{R}_-}(u) = \infty$ otherwise, which we approximate with a logarithmic barrier



$$\begin{array}{ll} \text{minimize} & f_0(x) - \frac{1}{t}\sum_{i=1}^{p} \log(-f_i(x)) \\ \text{subject to} & Ax = b \end{array}$$

to get an equality constrained approximation.

Algorithms for Large Scale Problems

- for large scale problems, e.g., deep learning, Newton's method is too expensive
- even computing the true gradient may be too expensive
- many loss functions in machine learning decompose over train data $\{(x_i, y_i)\}_{i=1}^m$,

$$L(\theta) = \sum_{i=1}^{m} \ell(f(x_i;\theta), y_i)$$

- SGD approximates the gradient on mini-batches $\mathcal{I} \subseteq \{1, \ldots, m\}$

$$\widehat{\nabla_\theta L} = \sum_{i \in \mathcal{I}} \nabla_\theta \ell(f(x_i;\theta), y_i)$$

- under mild assumptions $E\left[\widehat{\nabla_\theta L}\right] = \nabla_\theta L$
- for constrained problems can project back onto feasible set

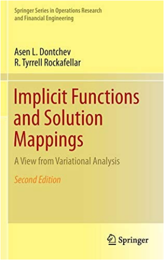
Many, many other schemes and variations!

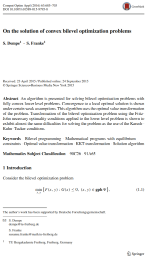
Non-convex Optimization

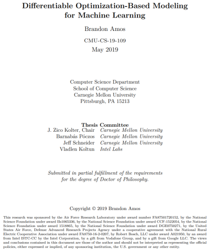
- deep learning (and many other problems) are non-convex
  - results from composition of architecture and loss function
  - need to make compromises
  - solvers often based on solving convex subproblems
- local optimization techniques
  - find a point that minimizes f_0 among all feasible points near it
  - requires good initial guess
  - no guarantees on distance to (global) optimum
- global optimization techniques
  - finds the true global solution
  - worse-case complexity grows exponentially with problem size
- we often only care about a *good* solution, not necessarily the optimal one

part 2

Part 2: Differentiable Optimisation and Deep Learning

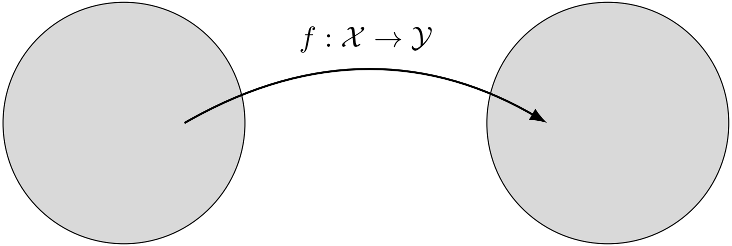




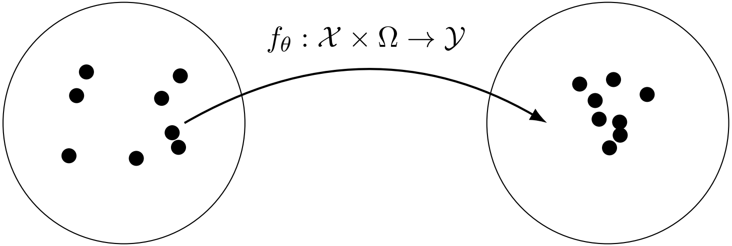


https://deepdeclarativenetworks.com

Machine Learning from 10,000ft



Machine Learning from 10,000ft

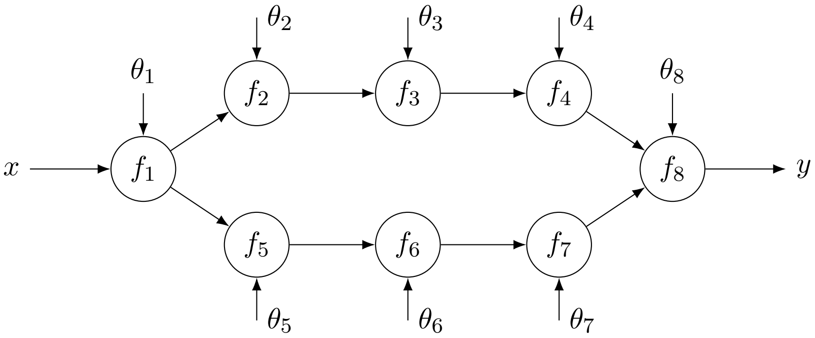


minimize (over θ) $\sum_{(x,y)\sim\mathcal{X}\times\mathcal{Y}} L(f_\theta(x), y)$

- loss L — what to do
- model f_θ — how to do it
- optimised by gradient descent

Deep Learning as an End-to-end Computation Graph

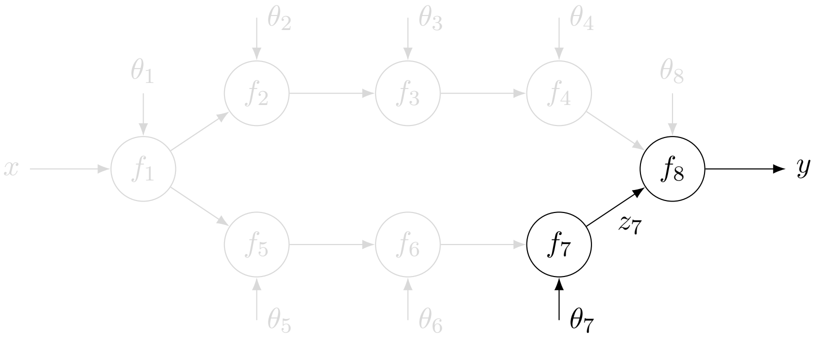
Deep learning does this by defining a function (equiv. computation graph) composed of many simple parametrized functions (equiv. computation nodes).



$$y = f_8(f_4(f_3(f_2(f_1(x)))), f_7(f_6(f_5(f_1(x)))))$$

(parameters θ_i omitted for brevity)

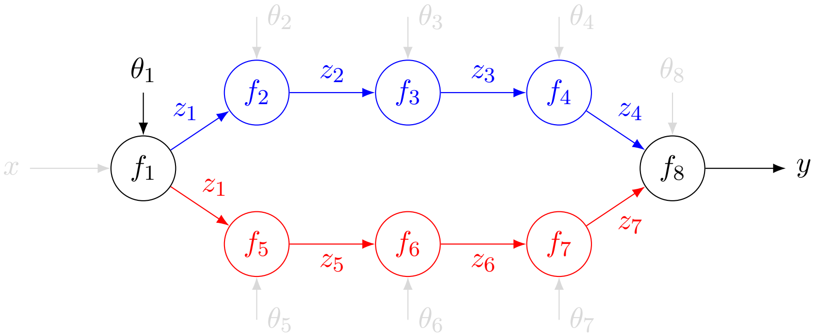
Backward Pass



Example 1.

$$\frac{\partial L}{\partial \theta_7} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial z_7}\frac{\partial z_7}{\partial \theta_7}$$

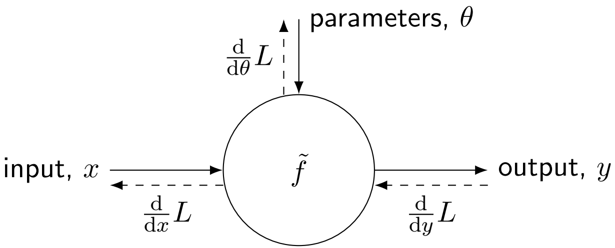
Backward Pass



Example 2.

$$\frac{\partial L}{\partial \theta_1} = \frac{\partial L}{\partial y}\left(\frac{\partial y}{\partial z_4}\frac{\partial z_4}{\partial z_3}\frac{\partial z_3}{\partial z_2}\frac{\partial z_2}{\partial z_1} + \frac{\partial y}{\partial z_7}\frac{\partial z_7}{\partial z_6}\frac{\partial z_6}{\partial z_5}\frac{\partial z_5}{\partial z_1}\right)\frac{\partial z_1}{\partial \theta_1}$$

Deep Learning Node



- **Forward pass:** compute output y as a function of the input x (and model parameters θ).
- **Backward pass:** compute the derivative of the loss with respect to the input x (and model parameters θ) given the derivative of the loss with respect to the output y.

Aside: Notation (Often Sloppy)

“the wonderful thing about standards is that there are so many of them to choose from”

For scalar-valued functions:

total derivative: $\dfrac{\mathrm{d}f}{\mathrm{d}x}$ partial derivative: $\dfrac{\partial f}{\partial x}$

For multi-dimensional scalar-valued functions, $f : \mathbb{R}^n \to \mathbb{R}$:

$$\nabla f(x) = \left(\frac{\mathrm{d}f}{\mathrm{d}x_1}, \ldots, \frac{\mathrm{d}f}{\mathrm{d}x_n} \right) \in \mathbb{R}^n$$

Aside: Notation (Often Sloppy)

"the wonderful thing about standards is that there are so many of them to choose from"

For scalar-valued functions:

total derivative: $\dfrac{\mathrm{d}f}{\mathrm{d}x}$ partial derivative: $\dfrac{\partial f}{\partial x}$

For multi-dimensional scalar-valued functions, $f : \mathbb{R}^n \to \mathbb{R}$:

$$\nabla f(x) = \left(\frac{\mathrm{d}f}{\mathrm{d}x_1}, \ldots, \frac{\mathrm{d}f}{\mathrm{d}x_n} \right) \in \mathbb{R}^n$$

Aside: Notation (Often Sloppy)

For scalar-valued functions:

total derivative: $\dfrac{\mathrm{d}f}{\mathrm{d}x}$ partial derivative: $\dfrac{\partial f}{\partial x}$

For multi-dimensional vector-valued functions, $f : \mathbb{R}^n \to \mathbb{R}^m$:

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = \begin{bmatrix} \frac{\mathrm{d}f_1}{\mathrm{d}x_1} & \cdots & \frac{\mathrm{d}f_1}{\mathrm{d}x_n} \\ \vdots & \ddots & \vdots \\ \frac{\mathrm{d}f_m}{\mathrm{d}x_1} & \cdots & \frac{\mathrm{d}f_m}{\mathrm{d}x_n} \end{bmatrix} \in \mathbb{R}^{m \times n} \qquad (\frac{\partial}{\partial x} f(x, y) \text{ for partial})$$

Sometimes D and D_X for $\frac{\mathrm{d}}{\mathrm{d}x}$ and $\frac{\partial}{\partial x}$, respectively.

Aside: Notation (Often Sloppy)

For scalar-valued functions:

total derivative: $\dfrac{\mathrm{d}f}{\mathrm{d}x}$ partial derivative: $\dfrac{\partial f}{\partial x}$

For multi-dimensional vector-valued functions, $f : \mathbb{R}^n \to \mathbb{R}^m$:

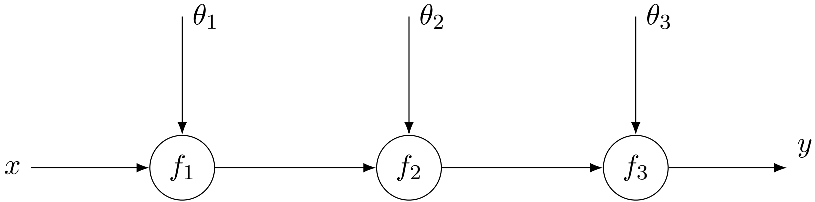
$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = \begin{bmatrix} \frac{\mathrm{d}f_1}{\mathrm{d}x_1} & \cdots & \frac{\mathrm{d}f_1}{\mathrm{d}x_n} \\ \vdots & \ddots & \vdots \\ \frac{\mathrm{d}f_m}{\mathrm{d}x_1} & \cdots & \frac{\mathrm{d}f_m}{\mathrm{d}x_n} \end{bmatrix} \in \mathbb{R}^{m \times n} \qquad (\frac{\partial}{\partial x} f(x, y) \text{ for partial})$$

Sometimes D and D_X for $\frac{\mathrm{d}}{\mathrm{d}x}$ and $\frac{\partial}{\partial x}$, respectively.

Mathematically, derivatives with respect to (scalar-valued) loss functions are row vectors ($m = 1$), i.e., $\nabla f(x)^T$.

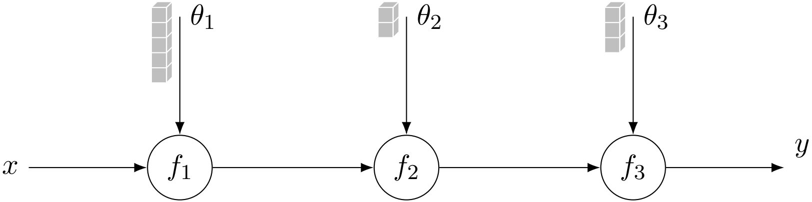
Concerning Memory

- data is often processed in batches ($B \times N \times \cdots \times C$)



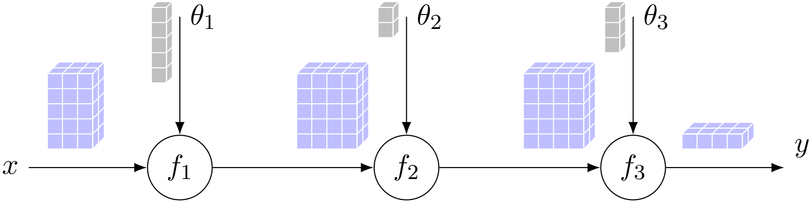
Concerning Memory

- data is often processed in batches ($B \times N \times \cdots \times C$)



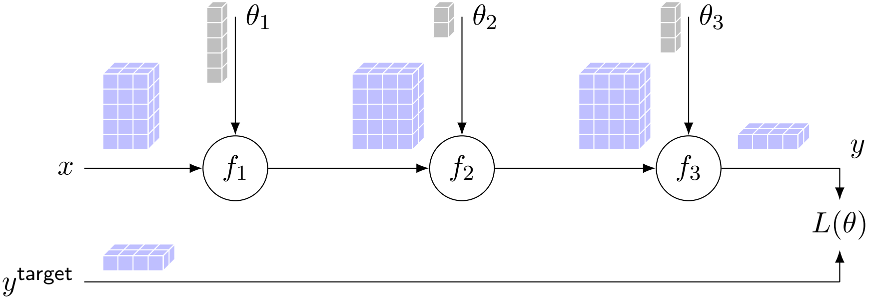
Concerning Memory

- data is often processed in batches ($B \times N \times \cdots \times C$)



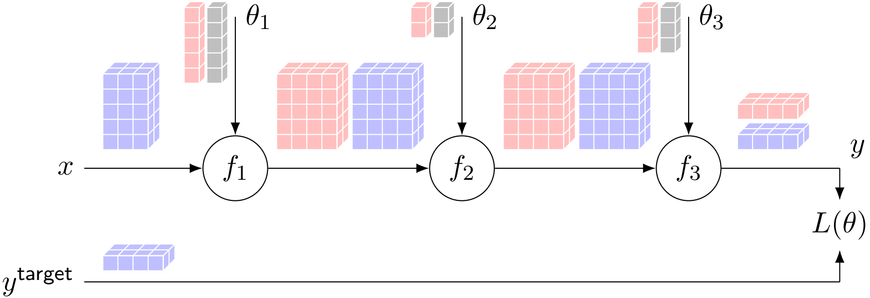
Concerning Memory

- data is often processed in batches ($B \times N \times \cdots \times C$)



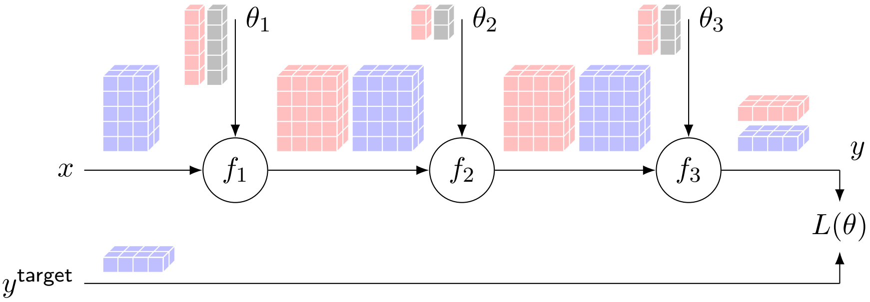
Concerning Memory

- data is often processed in batches $(B \times N \times \cdots \times C)$



Concerning Memory

- data is often processed in batches ($B \times N \times \cdots \times C$)

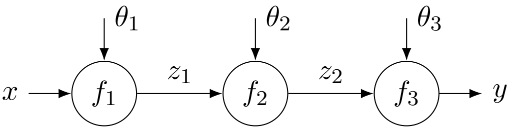


- parameters only take a small amount of memory (relative to data)
- gradients take the same amount of space as the data (stored transposed)
- in-place operations may save memory in the forward pass
- re-using buffers may save memory in the backward pass
- at test time intermediate results are not stored

Automatic Differentiation (AD)

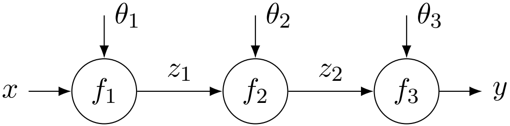
- algorithmic procedure that produces code for computing exact derivatives
- assumes numeric computations are composed of a small set of elementary operations that we know how to differentiate
  - arithmetic, exp, log, trigonometric
- workhorse of modern machine learning that greatly reduces development effort
- two flavours:
  - (forward mode) for a fixed independent variable u, computes derivatives $\frac{\mathrm{d}v}{\mathrm{d}u}$ for all dependent variables v
  - (reverse mode) for a fixed dependent variable v, computes derivatives $\frac{\mathrm{d}v}{\mathrm{d}u}$ for all independent variables u

Forward vs Reverse Mode Automatic Differentiation



$$\frac{\mathrm{d}L}{\mathrm{d}\theta_1} = \frac{\mathrm{d}L}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}z_2}\frac{\mathrm{d}z_2}{\mathrm{d}z_1}\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1}$$

Forward vs Reverse Mode Automatic Differentiation



$$\frac{\mathrm{d}L}{\mathrm{d}\theta_1} = \frac{\mathrm{d}L}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}z_2}\frac{\mathrm{d}z_2}{\mathrm{d}z_1}\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1}$$

Forward Mode

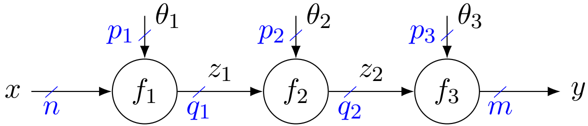
$$\frac{\mathrm{d}L}{\mathrm{d}\theta_1} = \frac{\mathrm{d}L}{\mathrm{d}y}\left(\frac{\mathrm{d}y}{\mathrm{d}z_2}\underbrace{\left(\frac{\mathrm{d}z_2}{\mathrm{d}z_1}\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1}\right)}_{\mathrm{d}z_2/\mathrm{d}\theta_1}\right)$$

Reverse Mode

$$\frac{\mathrm{d}L}{\mathrm{d}\theta_1} = \left(\underbrace{\left(\frac{\mathrm{d}L}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}z_2}\right)}_{\mathrm{d}L/\mathrm{d}z_2}\frac{\mathrm{d}z_2}{\mathrm{d}z_1}\right)\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1}$$

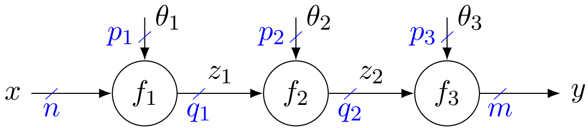
Cost of Gradient Evaluation Ordering

- in deep learning we usually want to update all parameters at the same time



Cost of Gradient Evaluation Ordering

- in deep learning we usually want to update all parameters at the same time

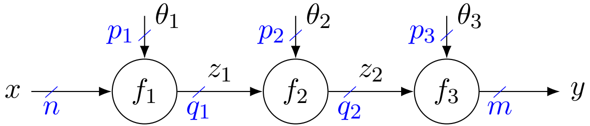


forward mode

$$\frac{\mathrm{d}L}{\mathrm{d}\theta_1} = \frac{\mathrm{d}L}{\mathrm{d}y}\,\frac{\mathrm{d}y}{\mathrm{d}z_2}\,\frac{\mathrm{d}z_2}{\mathrm{d}z_1}\,\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1}$$

Cost of Gradient Evaluation Ordering

- in deep learning we usually want to update all parameters at the same time

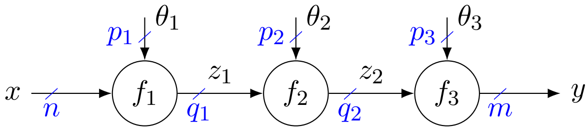


forward mode

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = \underbrace{\frac{\mathrm{d}L}{\mathrm{d}y}}_{1\times m} \underbrace{\frac{\mathrm{d}y}{\mathrm{d}z_2}}_{m\times q_2} \underbrace{\frac{\mathrm{d}z_2}{\mathrm{d}z_1}}_{q_2\times q_1} \underbrace{\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1}}_{q_1\times p_1}$$

Cost of Gradient Evaluation Ordering

- in deep learning we usually want to update all parameters at the same time

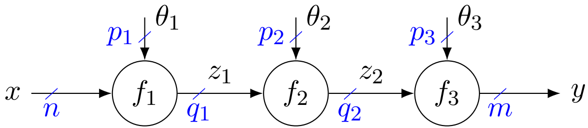


forward mode

$$\frac{\mathrm{d}L}{\mathrm{d}\theta_1} = \underbrace{\frac{\mathrm{d}L}{\mathrm{d}y}}_{1\times m}\underbrace{\frac{\mathrm{d}y}{\mathrm{d}z_2}}_{m\times q_2}\underbrace{\underbrace{\frac{\mathrm{d}z_2}{\mathrm{d}z_1}}_{q_2\times q_1}\underbrace{\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1}}_{q_1\times p_1}}_{q_2\times p_1}$$

Cost of Gradient Evaluation Ordering

- in deep learning we usually want to update all parameters at the same time

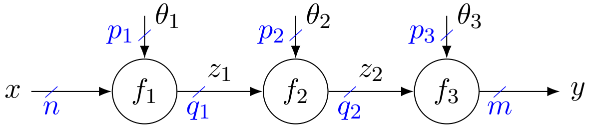


forward mode

$$\frac{\mathrm{d}L}{\mathrm{d}\theta_1} = \underbrace{\frac{\mathrm{d}L}{\mathrm{d}y}}_{1\times m} \underbrace{\underbrace{\frac{\mathrm{d}y}{\mathrm{d}z_2}}_{m\times q_2} \underbrace{\underbrace{\frac{\mathrm{d}z_2}{\mathrm{d}z_1}}_{q_2\times q_1} \underbrace{\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1}}_{q_1\times p_1}}_{q_2\times p_1}}_{m\times p_1}$$

Cost of Gradient Evaluation Ordering

- in deep learning we usually want to update all parameters at the same time

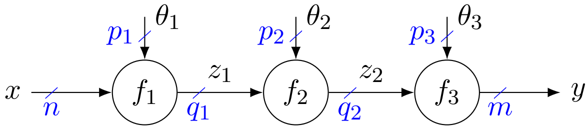


forward mode

$$\frac{\mathrm{d}L}{\mathrm{d}\theta_1} = \underbrace{\underbrace{\frac{\mathrm{d}L}{\mathrm{d}y}}_{1\times m} \underbrace{\underbrace{\frac{\mathrm{d}y}{\mathrm{d}z_2}}_{m\times q_2} \underbrace{\underbrace{\frac{\mathrm{d}z_2}{\mathrm{d}z_1}}_{q_2\times q_1} \underbrace{\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1}}_{q_1\times p_1}}_{q_2\times p_1}}_{m\times p_1}}_{1\times p_1}$$

Cost of Gradient Evaluation Ordering

- in deep learning we usually want to update all parameters at the same time



forward mode

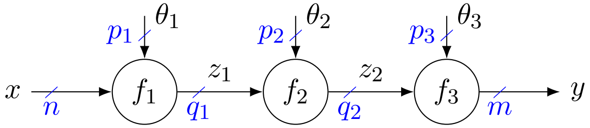
$$\frac{\mathrm{d}L}{\mathrm{d}\theta_1} = \underbrace{\underbrace{\frac{\mathrm{d}L}{\mathrm{d}y}}_{1\times m}\underbrace{\underbrace{\frac{\mathrm{d}y}{\mathrm{d}z_2}}_{m\times q_2}\underbrace{\underbrace{\frac{\mathrm{d}z_2}{\mathrm{d}z_1}}_{q_2\times q_1}\underbrace{\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1}}_{q_1\times p_1}}_{q_2\times p_1}}_{m\times p_1}}_{1\times p_1}$$

reverse mode

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = \underbrace{\underbrace{\underbrace{\underbrace{\frac{\mathrm{d}L}{\mathrm{d}y}}_{1\times m}\underbrace{\frac{\mathrm{d}y}{\mathrm{d}z_2}}_{m\times q_2}}_{1\times q_2}\underbrace{\frac{\mathrm{d}z_2}{\mathrm{d}z_1}}_{q_2\times q_1}}_{1\times q_1}\underbrace{\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1}}_{q_1\times p_1}}_{1\times p_1}$$

Cost of Gradient Evaluation Ordering

- in deep learning we usually want to update all parameters at the same time



forward mode

$$\frac{\mathrm{d}L}{\mathrm{d}\theta_1} = \underbrace{\underbrace{\frac{\mathrm{d}L}{\mathrm{d}y}}_{1\times m}\underbrace{\underbrace{\frac{\mathrm{d}y}{\mathrm{d}z_2}}_{m\times q_2}\underbrace{\underbrace{\frac{\mathrm{d}z_2}{\mathrm{d}z_1}}_{q_2\times q_1}\underbrace{\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1}}_{q_1\times p_1}}_{q_2\times p_1}}_{m\times p_1}}_{1\times p_1}$$

reverse mode

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = \underbrace{\underbrace{\underbrace{\underbrace{\frac{\mathrm{d}L}{\mathrm{d}y}}_{1\times m}\underbrace{\frac{\mathrm{d}y}{\mathrm{d}z_2}}_{m\times q_2}}_{1\times q_2}\underbrace{\frac{\mathrm{d}z_2}{\mathrm{d}z_1}}_{q_2\times q_1}}_{1\times q_1}\underbrace{\frac{\mathrm{d}z_1}{\mathrm{d}\theta_1}}_{q_1\times p_1}}_{1\times p_1}$$

- reverse mode operations are called vector-Jacobian products
- more efficient for deep learning (scalar L, vector θ) than forward mode

Back-propagation

- reverse mode AD is called **back-propagation** in deep learning
- different deep learning frameworks use slightly different approaches (explicit graph construction versus implicit operator tracking)
- conceptually, for every line of code:

```
P, Q = foo(A, B, C)
```

  automatic differentiation produces line:

```
dLdA, dLdB, dLdC = foo_vjp(A, B, C, P, Q, dLdP, dLdQ)
```

- requires two passes through the network (one forward, one backward)

Toy Example: Babylonian Algorithm

Consider the following implementation for a forward operation:

procedure $\textsc{FwdFcn}(x)$
  $y_0 \leftarrow \frac{1}{2}x$
  for $t = 1, \ldots, T$ **do**
    $y_t \leftarrow \frac{1}{2}\left(y_{t-1} + \frac{x}{y_{t-1}}\right)$
  end for
  return y_T
end procedure

Toy Example: Babylonian Algorithm

Consider the following implementation for a forward operation:

procedure $\textsc{FwdFcn}(x)$
    $y_0 \leftarrow \frac{1}{2}x$
    for $t = 1, \ldots, T$ **do**
        $y_t \leftarrow \frac{1}{2}\left(y_{t-1} + \frac{x}{y_{t-1}}\right)$
    end for
    return y_T
end procedure

Automatic differentiation algorithmically generates the backward code:

procedure $\textsc{BckFcn}(x, y_T, \frac{\mathrm{d}L}{\mathrm{d}y_T})$
    $\frac{\mathrm{d}L}{\mathrm{d}x} \leftarrow 0$
    for $t = T, \ldots, 1$ **do**
        $\frac{\mathrm{d}L}{\mathrm{d}x} \leftarrow \frac{\mathrm{d}L}{\mathrm{d}x} + \frac{\mathrm{d}L}{\mathrm{d}y_t} \overbrace{\left(\frac{1}{2y_{t-1}}\right)}^{\partial y_t / \partial x}$
        $\frac{\mathrm{d}L}{\mathrm{d}y_{t-1}} \leftarrow \frac{\mathrm{d}L}{\mathrm{d}y_t} \underbrace{\left(\frac{1}{2} - \frac{x}{2y_{t-1}^2}\right)}_{\partial y_t / \partial y_{t-1}}$
    end for
    $\frac{\mathrm{d}L}{\mathrm{d}x} \leftarrow \frac{\mathrm{d}L}{\mathrm{d}x} + \frac{\mathrm{d}L}{\mathrm{d}y_0} \frac{1}{2}$
    return $\frac{\mathrm{d}L}{\mathrm{d}x}$
end procedure

Toy Example: Babylonian Algorithm

Consider the following implementation for a forward operation:

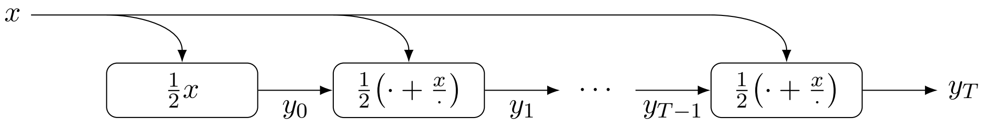
procedure $\textsc{FwdFcn}(x)$
  $y_0 \leftarrow \frac{1}{2}x$
  for $t = 1, \dots, T$ **do**
    $y_t \leftarrow \frac{1}{2}\left(y_{t-1} + \frac{x}{y_{t-1}}\right)$
  end for
  return y_T
end procedure

- computes $y = \sqrt{x}$
- derivative could be computed directly as $\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{1}{2\sqrt{x}} = \frac{1}{2y}$

Automatic differentiation algorithmically generates the backward code:

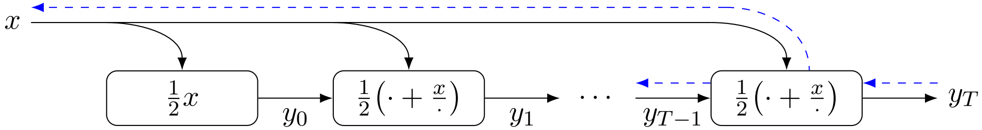
procedure $\textsc{BckFcn}(x, y_T, \frac{\mathrm{d}L}{\mathrm{d}y_T})$
  $\frac{\mathrm{d}L}{\mathrm{d}x} \leftarrow 0$
  for $t = T, \dots, 1$ **do**
    $\frac{\mathrm{d}L}{\mathrm{d}x} \leftarrow \frac{\mathrm{d}L}{\mathrm{d}x} + \frac{\mathrm{d}L}{\mathrm{d}y_t} \overbrace{\left(\frac{1}{2y_{t-1}}\right)}^{\partial y_t / \partial x}$
    $\frac{\mathrm{d}L}{\mathrm{d}y_{t-1}} \leftarrow \frac{\mathrm{d}L}{\mathrm{d}y_t} \underbrace{\left(\frac{1}{2} - \frac{x}{2y_{t-1}^2}\right)}_{\partial y_t / \partial y_{t-1}}$
  end for
  $\frac{\mathrm{d}L}{\mathrm{d}x} \leftarrow \frac{\mathrm{d}L}{\mathrm{d}x} + \frac{\mathrm{d}L}{\mathrm{d}y_0} \frac{1}{2}$
  return $\frac{\mathrm{d}L}{\mathrm{d}x}$
end procedure

Computation Graph for Babylonian Algorithm



$$y_T = f(x, f(x, f(x, \dots f(x, \tfrac{1}{2}x)))) \text{ with } f(x, y) = \frac{1}{2}\left(y + \frac{x}{y}\right)$$

Computation Graph for Babylonian Algorithm



$$y_T = f(x, f(x, f(x, \dots f(x, \tfrac{1}{2}x)))) \text{ with } f(x,y) = \frac{1}{2}\left(y + \frac{x}{y}\right)$$

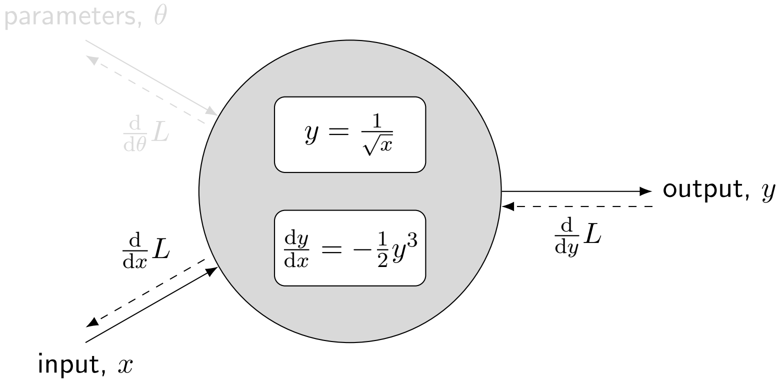
$$\frac{\mathrm{d}L}{\mathrm{d}x} = \frac{\mathrm{d}L}{\mathrm{d}y_T}\left(\frac{\partial y_T}{\partial x} + \frac{\partial y_T}{\partial y_{T-1}}\left(\frac{\partial y_{T-1}}{\partial x} + \frac{\partial y_{T-1}}{\partial y_{T-2}}\left(\dots + \frac{\partial y_0}{\partial x}\right)\right)\right)$$

but automatic differentiation doesn’t always work...

Computing $1/\sqrt{x}$

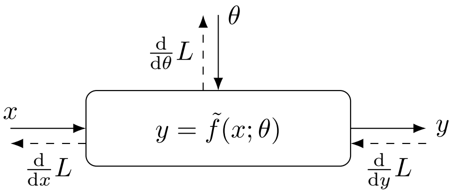
```
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = * ( long * ) &y;                  // evil floating point bit level hacking
    i = 0x5f3759df - ( i >> 1 );          // what the f**k?
    y = * ( float * ) &i;
    y = y * ( threehalfs - ( x2 * y * y ) );      // 1st iter
    //  y = y * ( threehalfs - ( x2 * y * y ) ); // 2nd iter, can be removed

    return y;
}
```

Separate Forward and Backward Operations



embedding optimisation problems inside deep learning networks

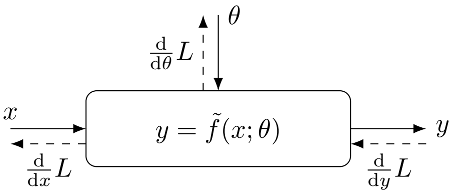
Imperative vs Declarative Nodes

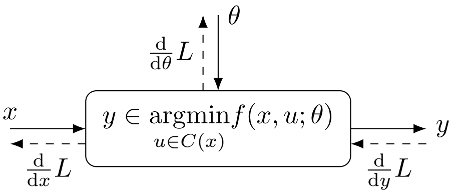


- imperative node
- input-output relationship explicit,

$$y = \tilde{f}(x;\theta)$$

Imperative vs Declarative Nodes





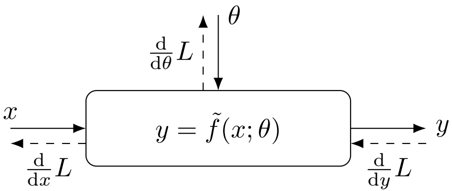
- imperative node
- input-output relationship explicit,

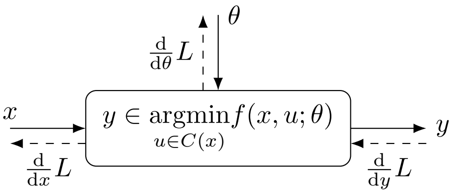
$$y = \tilde{f}(x;\theta)$$

- declarative node
- input-output relationship specified as solution to an optimisation problem,

$$y \in \arg\min_{u \in C(x)} f(x, u; \theta)$$

Imperative vs Declarative Nodes





- imperative node
- input-output relationship explicit,

$$y = \tilde{f}(x;\theta)$$

- declarative node
- input-output relationship specified as solution to an optimisation problem,

$$y \in \arg\min_{u \in C(x)} f(x, u; \theta)$$

can co-exist in the same computation graph (network)

Average Pooling Example

$$\{x_i \in \mathbb{R}^m \mid i = 1, \ldots, n\} \to \mathbb{R}^m$$

- imperative specification

$$y = \frac{1}{n} \sum_{i=1}^{n} x_i$$

- declarative specification

$$y = \text{argmin}_{u \in \mathbb{R}^m} \sum_{i=1}^{n} \|u - x_i\|^2$$

Average Pooling Example

$$\{x_i \in \mathbb{R}^m \mid i = 1, \ldots, n\} \to \mathbb{R}^m$$

- imperative specification

$$y = \frac{1}{n} \sum_{i=1}^{n} x_i$$

- declarative specification

$$y = \text{argmin}_{u \in \mathbb{R}^m} \sum_{i=1}^{n} \|u - x_i\|^2$$

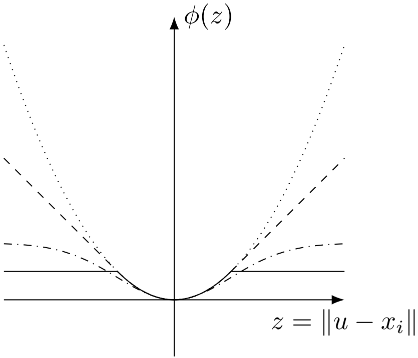
- can be easily varied, e.g., made robust

$$y = \text{argmin}_{u \in \mathbb{R}^m} \sum_{i=1}^{n} \phi(u - x_i)$$

for some penalty function ϕ

Average Pooling Example

$$\{x_i \in \mathbb{R}^m \mid i = 1, \ldots, n\} \to \mathbb{R}^m$$



- declarative specification

$$y = \text{argmin}_{u \in \mathbb{R}^m} \sum_{i=1}^{n} \|u - x_i\|^2$$

- can be easily varied, e.g., made robust

$$y = \text{argmin}_{u \in \mathbb{R}^m} \sum_{i=1}^{n} \phi(u - x_i)$$

for some penalty function ϕ

Bi-level Optimisation: Stackelberg Games

Consider two players, a **leader** and a **follower**

- the market dictates the price it's willing to pay for some goods based on supply, i.e., quantity produced by both players, $P(q_1 + q_2)$
- each player has a cost structure associated with producing goods, $C_i(q_i)$ and wants to maximize profits, $q_i P(q_1 + q_2) - C_i(q_i)$
- the leader picks a quantity of goods to produce knowing that the follower will respond optimally. In other words, the leader solves

$$\begin{array}{ll} \text{maximize (over } q_1) & q_1 P(q_1 + q_2) - C_1(q_1) \\ \text{subject to} & q_2 \in \operatorname{argmax}_q \, q P(q_1 + q) - C_2(q) \end{array}$$

Solving Bi-level Optimisation Problems

$$
\begin{array}{ll}
\text{minimize (over } x\text{)} & L(x, y) \\
\text{subject to} & y \in \operatorname{argmin}_{u \in C(x)} f(x, u)
\end{array}
$$

Solving Bi-level Optimisation Problems

$$\begin{array}{ll} \text{minimize (over } x\text{)} & L(x, y) \\ \text{subject to} & y \in \operatorname{argmin}_{u \in C(x)} f(x, u) \end{array}$$

- **closed-form solution:** substitute for y in upper-level problem (if possible)

$$\text{minimize (over } x\text{)} \quad L(x, y(x))$$

Solving Bi-level Optimisation Problems

$$\begin{array}{ll} \text{minimize (over } x\text{)} & L(x, y) \\ \text{subject to} & y \in \operatorname{argmin}_{u \in C(x)} f(x, u) \end{array}$$

- **closed-form solution:** substitute for y in upper-level problem (if possible)

$$\text{minimize (over } x\text{)} \quad L(x, y(x))$$

- **convex lower-level problem:** replace lower-level problem with sufficient optimality conditions (e.g., KKT conditions),

$$\begin{array}{ll} \text{minimize (over } x, y\text{)} & L(x, y) \\ \text{subject to} & h(x, y) = 0 \end{array}$$

Solving Bi-level Optimisation Problems

$$\begin{array}{ll} \text{minimize (over } x) & L(x, y) \\ \text{subject to} & y \in \operatorname{argmin}_{u \in C(x)} f(x, u) \end{array}$$

- **closed-form solution:** substitute for y in upper-level problem (if possible)

$$\text{minimize (over } x) \quad L(x, y(x))$$

- **convex lower-level problem:** replace lower-level problem with sufficient optimality conditions (e.g., KKT conditions),

$$\begin{array}{ll} \text{minimize (over } x, y) & L(x, y) \\ \text{subject to} & h(x, y) = 0 \end{array}$$

- **gradient descent:** compute gradient of lower-level solution y with respect to x, and use the chain rule to get the total derivative,

$$x \leftarrow x - \eta \left(\frac{\partial L(x, y)}{\partial x} + \frac{\partial L(x, y)}{\partial y} \frac{\mathrm{d}y}{\mathrm{d}x} \right)$$

Solving Bi-level Optimisation Problems

$$\begin{array}{ll} \text{minimize (over } x) & L(x, y) \\ \text{subject to} & y \in \operatorname{argmin}_{u \in C(x)} f(x, u) \end{array}$$

- **closed-form solution:** substitute for y in upper-level problem (if possible)

$$\text{minimize (over } x) \quad L(x, y(x))$$

- **convex lower-level problem:** replace lower-level problem with sufficient optimality conditions (e.g., KKT conditions),

$$\begin{array}{ll} \text{minimize (over } x, y) & L(x, y) \\ \text{subject to} & h(x, y) = 0 \end{array}$$

- **gradient descent:** compute gradient of lower-level solution y with respect to x, and use the chain rule to get the total derivative,

$$x \leftarrow x - \eta \left(\frac{\partial L(x, y)}{\partial x} + \frac{\partial L(x, y)}{\partial y} \frac{\mathrm{d}y}{\mathrm{d}x} \right)$$

  - by back-propagating through optimisation procedure or implicit differentiation

Differentiable Least Squares

Consider our old friend, the least-squares problem,

$$\text{minimize} \quad \|Ax - b\|_2^2$$

parameterized by A and b and with closed-form solution $x^\star = \left(A^T A\right)^{-1} A^T b$.

Differentiable Least Squares

Consider our old friend, the least-squares problem,

$$\text{minimize} \quad \|Ax - b\|_2^2$$

parameterized by A and b and with closed-form solution $x^\star = \left(A^T A\right)^{-1} A^T b$.

We are interested in derivatives of the solution with respect to the elements of A,

$$\frac{\mathrm{d}x^\star}{\mathrm{d}A_{ij}} = \frac{\mathrm{d}}{\mathrm{d}A_{ij}} \left(A^T A\right)^{-1} A^T b \quad \in \mathbb{R}^n$$

We could also compute derivatives with respect to elements of b (but not here).

Least Squares Backward Pass

The backward pass combines $\frac{\mathrm{d}x^\star}{\mathrm{d}A_{ij}}$ with $v^T = \frac{\mathrm{d}L}{\mathrm{d}x^\star}$ via the vector-Jacobian product. After some algebraic manipulation (see lecture notes) we get

$$\left(\frac{\mathrm{d}L}{\mathrm{d}A}\right)^T = wr^T - x^\star(Aw)^T \quad \in \mathbb{R}^{m\times n}$$

where $w^T = v^T(A^TA)^{-1}$ and $r = b - Ax^\star$.

Least Squares Backward Pass

The backward pass combines $\frac{\mathrm{d}x^\star}{\mathrm{d}A_{ij}}$ with $v^T = \frac{\mathrm{d}L}{\mathrm{d}x^\star}$ via the vector-Jacobian product. After some algebraic manipulation (see lecture notes) we get

$$\left(\frac{\mathrm{d}L}{\mathrm{d}A}\right)^T = wr^T - x^\star(Aw)^T \quad \in \mathbb{R}^{m\times n}$$

where $w^T = v^T(A^TA)^{-1}$ and $r = b - Ax^\star$.

- $\left(A^TA\right)^{-1}$ is used in both the forward and backward pass
- factored once to solve for x, e.g., into $A = QR$
- cache R and re-use when computing gradients

PyTorch Implementation: Forward Pass

```python
class LeastSquaresFcn(torch.autograd.Function):
    """PyTorch autograd function for least squares."""

    @staticmethod
    def forward(ctx, A, b):
        B, M, N = A.shape
        assert b.shape == (B, M, 1)

        with torch.no_grad():
        Q, R = torch.linalg.qr(A, mode='reduced')
        x = torch.linalg.solve_triangular(R,
            torch.bmm(b.view(B, 1, M), Q).view(B, N, 1), upper=True)

        # save state for backward pass
        ctx.save_for_backward(A, b, x, R)

        # return solution
        return x
```

$$A = QR$$

$$x = R^{-1}\left(Q^T b\right)$$

(solves $Rx = Q^T b$)

PyTorch Implementation: Backward Pass

```python
    @staticmethod
    def backward(ctx, dx):
        # check for None tensors
        if dx is None:
            return None, None

        # unpack cached tensors
        A, b, x, R = ctx.saved_tensors
        B, M, N = A.shape

        dA, db = None, None

        w = torch.linalg.solve_triangular(R,
            torch.linalg.solve_triangular(torch.transpose(R, 2, 1),
            dx, upper=False), upper=True)
        Aw = torch.bmm(A, w)

        if ctx.needs_input_grad[0]:
            r = b - torch.bmm(A, x)
            dA = torch.bmm(r.view(B,M,1), w.view(B,1,N)) - \
                torch.bmm(Aw.view(B,M,1), x.view(B,1,N))
        if ctx.needs_input_grad[1]:
            db = Aw

        # return gradients
        return dA, db
```

$$\begin{aligned} w &= \left(A^T A\right)^{-1} v \\ &= R^{-1}\left(R^{-T} v\right) \end{aligned}$$

$$r = b - Ax$$

$$\left(\frac{\mathrm{d}L}{\mathrm{d}A}\right)^T = rw^T - (Aw)x^T$$

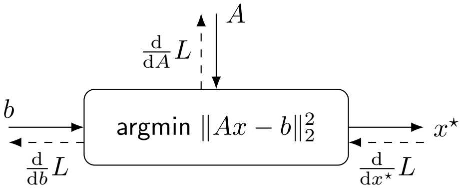
$$\left(\frac{\mathrm{d}L}{\mathrm{d}b}\right)^T = Aw$$

Example

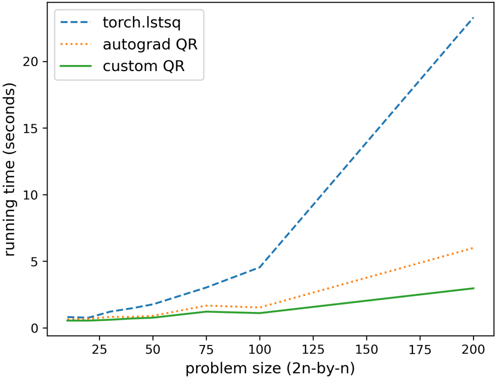
Bi-level optimisation problem with lower-level least squares:

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\|x^\star - x^{\text{target}}\|_2^2 \\ \text{subject to} & x^\star = \operatorname{argmin}_x \|Ax - b\|_2^2 \end{array}$$

with upper-level variable $A \in \mathbb{R}^{m \times n}$.

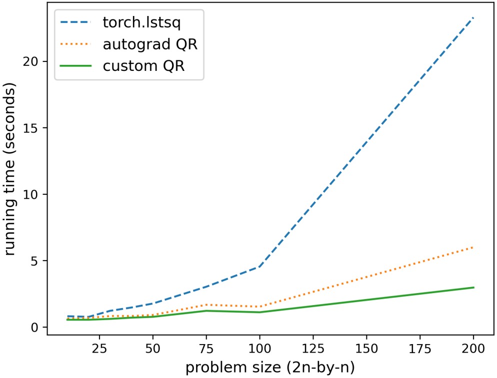


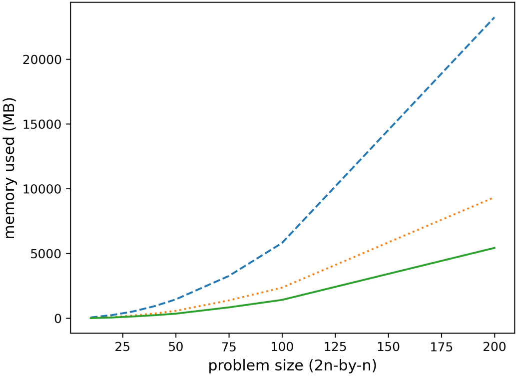
Profiling



(problems with $m = 2n$; run for 1000 iterations on CPU using PyTorch 1.13.0)

Profiling





(problems with $m = 2n$; run for 1000 iterations on CPU using PyTorch 1.13.0)

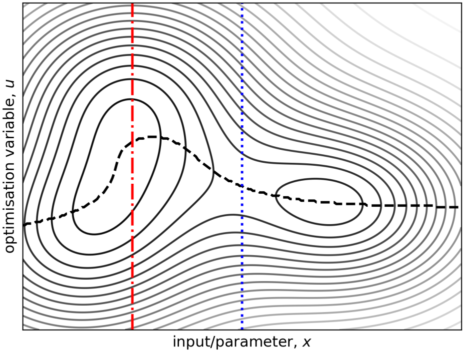
Parametrized Optimisation

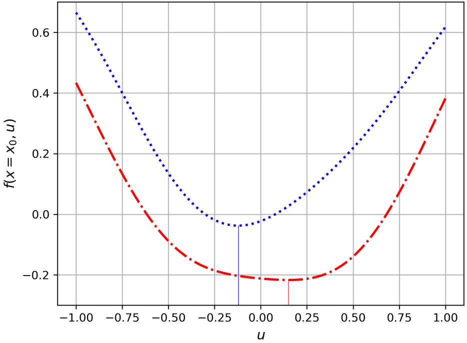
In the context of deep learning the upper-level Stackelberg problem is the **learning problem** and the lower-level Stackelberg problem is the **inference problem**.

A declarative node defines a family of problems indexed by continuous variable $x \in \mathbb{R}^n$,

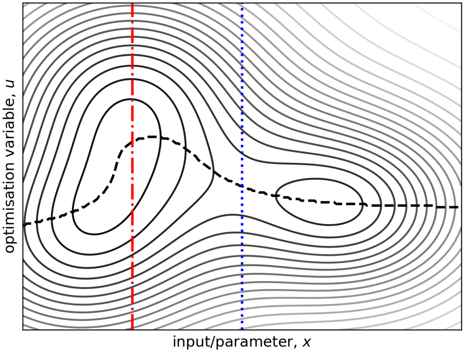
$$\left\{ \begin{array}{lll} \text{minimize (over } u \in \mathbb{R}^m) & f_0(x, u) & \\ \text{subject to} & f_i(x, u) \leq 0, & i = 1, \ldots, p \\ & h_i(x, u) = 0, & i = 1, \ldots, q \end{array} \right\}_{x \in \mathbb{R}^n}$$

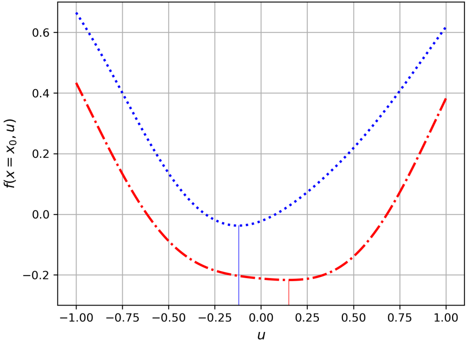
Bi-level Optimisation Ambient Space





Bi-level Optimisation Ambient Space





Main question: How do we compute $\frac{\mathrm{d}}{\mathrm{d}x} \operatorname{argmin}_{u \in C} f(x, u)$ if we don't have a closed-form solution?

Unrolling Iterations

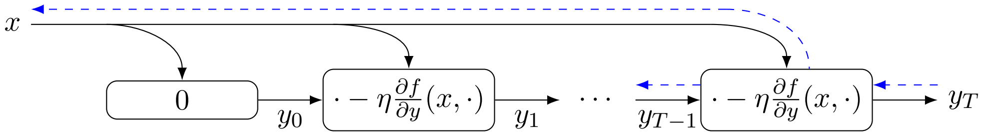
Automatic differentiation can be applied to back-propagate through iterative algorithms such as gradient descent for unconstrained optimisation.

- we saw this in the Babylonian algorithm example

Unrolling Iterations

Automatic differentiation can be applied to back-propagate through iterative algorithms such as gradient descent for unconstrained optimisation.

- we saw this in the Babylonian algorithm example

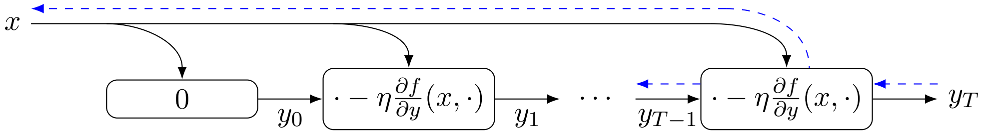


- **forward pass:** $y_t \leftarrow y_{t-1} - \eta \frac{\partial f}{\partial y}(x, y_{t-1})$
- **backward pass:** $\frac{\mathrm{d}y_t}{\mathrm{d}x} = \underbrace{-\eta \frac{\partial^2 f}{\partial x \partial y}(x, y_{t-1})}_{\frac{\partial y_t}{\partial x}} + \underbrace{\left(I - \eta \frac{\partial^2 f}{\partial y^2}(x, y_{t-1})\right)}_{\frac{\partial y_t}{\partial y_{t-1}}} \frac{\mathrm{d}y_{t-1}}{\mathrm{d}x}$

Unrolling Iterations

Automatic differentiation can be applied to back-propagate through iterative algorithms such as gradient descent for unconstrained optimisation.

- we saw this in the Babylonian algorithm example



- **forward pass:** $y_t \leftarrow y_{t-1} - \eta \frac{\partial f}{\partial y}(x, y_{t-1})$
- **backward pass:** $\frac{\mathrm{d}y_t}{\mathrm{d}x} = -\eta \frac{\partial^2 f}{\partial x \partial y}(x, y_{t-1}) + \left(I - \eta \frac{\partial^2 f}{\partial y^2}(x, y_{t-1})\right) \frac{\mathrm{d}y_{t-1}}{\mathrm{d}x}$

But as we will see, using the idea of separate forward and backward pass code, there is a much better way to do this by computing $\frac{\mathrm{d}y}{\mathrm{d}x}$ directly.

Dini's Implicit Function Theorem

Consider the solution mapping associated with the equation $\zeta(x, u) = 0$,

$$Y : x \mapsto \{u \in \mathbb{R}^m \mid \zeta(x, u) = 0\} \text{ for } x \in \mathbb{R}^n.$$

We are interested in how elements of $Y(x)$ change as a function of x.

Dini's Implicit Function Theorem

Consider the solution mapping associated with the equation $\zeta(x, u) = 0$,

$$Y : x \mapsto \{u \in \mathbb{R}^m \mid \zeta(x, u) = 0\} \text{ for } x \in \mathbb{R}^n.$$

We are interested in how elements of $Y(x)$ change as a function of x.

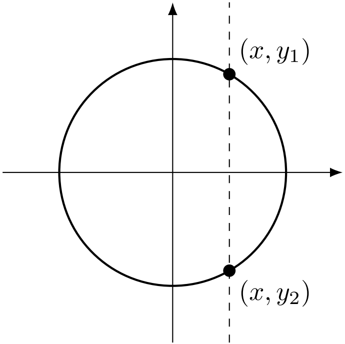
Theorem

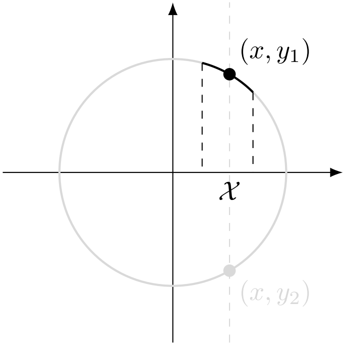
Let $\zeta : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$ be differentiable in a neighbourhood of (x, u) and such that $\zeta(x, u) = 0$, and let $\frac{\partial}{\partial u}\zeta(x, u)$ be nonsingular. Then the solution mapping Y has a single-valued localization y around x for u which is differentiable in a neighbourhood $\mathcal{X}$ of x with Jacobian satisfying

$$\frac{dy(x)}{dx} = -\left(\frac{\partial \zeta(x, y(x))}{\partial y}\right)^{-1} \frac{\partial \zeta(x, y(x))}{\partial x}$$

for every $x \in \mathcal{X}$.

Unit Circle Example





$$y = \pm\sqrt{1 - x^2}$$
$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mp 2x}{2\sqrt{1 - x^2}} = -\frac{x}{y}$$

$$\zeta(x, y) = x^2 + y^2 - 1$$
$$\frac{\mathrm{d}y}{\mathrm{d}x} = -\left(\frac{\partial \zeta}{\partial y}\right)^{-1}\left(\frac{\partial \zeta}{\partial x}\right)$$
$$= -\left(\frac{1}{2y}\right)(2x) = -\frac{x}{y}$$

Differentiating Unconstrained Optimisation Problems

Let $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ be twice differentiable and let

$$y(x) \in \text{argmin}_u f(x, u)$$

then for non-zero Hessian

$$\frac{\mathrm{d}y(x)}{\mathrm{d}x} = -\left(\frac{\partial^2 f}{\partial y^2}\right)^{-1} \frac{\partial^2 f}{\partial x \partial y}.$$

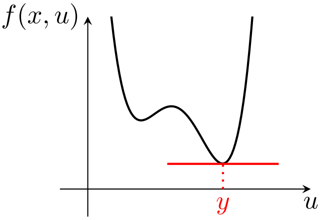
Differentiating Unconstrained Optimisation Problems

Let $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ be twice differentiable and let

$$y(x) \in \operatorname{argmin}_u f(x,u)$$

then for non-zero Hessian

$$\frac{\mathrm{d}y(x)}{\mathrm{d}x} = -\left(\frac{\partial^2 f}{\partial y^2}\right)^{-1} \frac{\partial^2 f}{\partial x \partial y}.$$



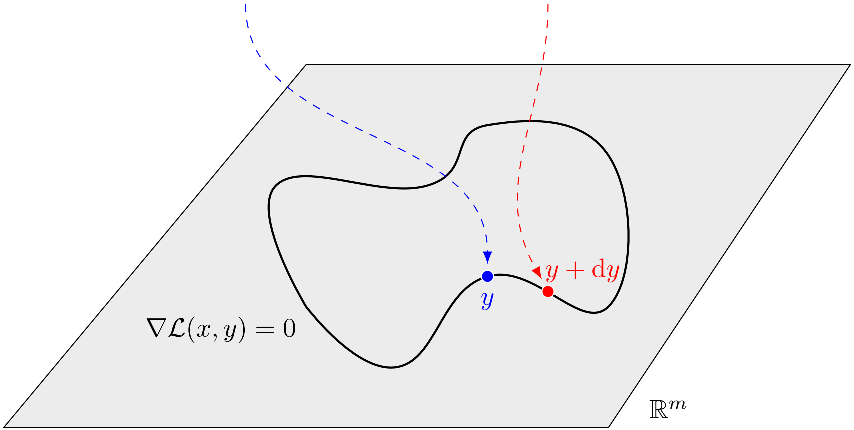
Proof. The derivative of f vanishes at (x,y), i.e., $y \in \operatorname{argmin}_u f(x,u) \implies \frac{\partial f(x,y)}{\partial y} = 0$.

$$\begin{aligned}
\text{LHS :} \quad & \frac{\mathrm{d}}{\mathrm{d}x}\frac{\partial f(x,y)}{\partial y} && = \frac{\partial^2 f(x,y)}{\partial x \partial y} + \frac{\partial^2 f(x,y)}{\partial y^2}\frac{\mathrm{d}y}{\mathrm{d}x} \\
\text{RHS :} \quad & \frac{\mathrm{d}}{\mathrm{d}x}0 && = 0
\end{aligned}$$

Equating and rearranging gives the result. Or directly from Dini's implicit function theorem on $\underbrace{\frac{\partial f(x,y)}{\partial y}}_{\zeta(x,y)} = 0$.

Differentiable Optimisation: Big Picture Idea

$$\begin{array}{rl} \text{min.} & f_0(x, u) \\ \text{s.t.} & u \in C(x) \end{array} \qquad\qquad \begin{array}{rl} \text{min.} & f_0(x + \mathrm{d}x, u) \\ \text{s.t.} & u \in C(x + \mathrm{d}x) \end{array}$$



Differentiating Equality Constrained Optimisation Problems

Consider functions $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ and $h : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^q$. Let

$$\begin{array}{rll} y(x) \in & \arg\min_{u \in \mathbb{R}^m} & f(x, u) \\ & \text{subject to} & h(x, u) = 0_q \end{array}$$

Assume that $y(x)$ exists, that f and h are twice differentiable in the neighbourhood of $(x, y(x))$, and that **rank**$(\frac{\partial h(x,y)}{\partial y}) = q$.

Differentiating Equality Constrained Optimisation Problems

Consider functions $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ and $h : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^q$. Let

$$\begin{array}{rll} y(x) \in & \arg\min_{u \in \mathbb{R}^m} & f(x, u) \\ & \text{subject to} & h(x, u) = 0_q \end{array}$$

Assume that $y(x)$ exists, that f and h are twice differentiable in the neighbourhood of $(x, y(x))$, and that **rank**$(\frac{\partial h(x,y)}{\partial y}) = q$. Then for H non-singular

$$\frac{\mathrm{d}y(x)}{\mathrm{d}x} = H^{-1}A^T\left(AH^{-1}A^T\right)^{-1}\left(AH^{-1}B - C\right) - H^{-1}B$$

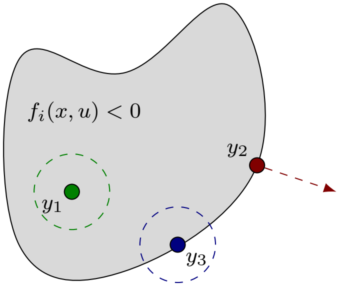
where

$$\begin{array}{ll} A = \frac{\partial h(x,y)}{\partial y} \in \mathbb{R}^{q \times m} & B = \frac{\partial^2 f(x,y)}{\partial x \partial y} - \sum_{i=1}^{q} \nu_i \frac{\partial^2 h_i(x,y)}{\partial x \partial y} \in \mathbb{R}^{m \times n} \\ C = \frac{\partial h(x,y)}{\partial x} \in \mathbb{R}^{q \times n} & H = \frac{\partial^2 f(x,y)}{\partial y^2} - \sum_{i=1}^{q} \nu_i \frac{\partial^2 h_i(x,y)}{\partial y^2} \in \mathbb{R}^{m \times m} \end{array}$$

and $\nu \in \mathbb{R}^q$ satisfies $\nu^T A = \frac{\partial f(x,y)}{\partial y}$.

Dealing with Inequality Constraints

$$
\begin{array}{lll}
y(x) \in \operatorname{arg\,min}_{u \in \mathbb{R}^m} & f_0(x, u) & \\
\text{subject to} & h_i(x, u) = 0, & i = 1, \ldots, q \\
& f_i(x, u) \leq 0, & i = 1, \ldots, p.
\end{array}
$$



- Replace inequality constraints with log-barrier approximation (see last lecture)
- Treat as equality constraints if active (y_2 or y_3) and ignore otherwise (y_1 or y_3)
  - may lead to one-sided gradients since $\lambda \succeq 0$

Automatic Differentiation for Differentiable Optimisation

- At one extreme we can try back propagate through the optimisation algorithm (i.e., unrolling the optimisation procedure using automatic differentiation)
- At the other extreme we can use the implicit differentiation result to hand-craft efficient backward pass code
- There are two options in between:
  - Use automatic differentiation to obtain quantities A, B, C and H from software implementations of the objective and (active) constraint functions
  - Implement the optimality condition $\nabla \mathcal{L} = 0$ in software and automatically differentiate that

Vector-Jacobian Product

For brevity consider the unconstrained optimisation case. The backward pass computes

$$\begin{aligned}\frac{\mathrm{d}L}{\mathrm{d}x} &= \color{red}{\frac{\mathrm{d}L}{\mathrm{d}y}}\color{blue}{\frac{\mathrm{d}y}{\mathrm{d}x}} \\ &= \underbrace{\color{red}{\left(v^T\right)}}_{\mathbb{R}^{1\times m}}\underbrace{\color{blue}{\left(-H^{-1}B\right)}}_{\mathbb{R}^{m\times n}}\end{aligned}$$

evaluation order:	$-v^T\left(H^{-1}B\right)$	$\left(-v^T H^{-1}\right)B$
cost†:	$O(m^2 n + mn)$	$O(m^2 + mn)$

† assumes H^{-1} is already factored (in $O(m^3)$ if unstructured, less if structured)

Summary and Open Questions

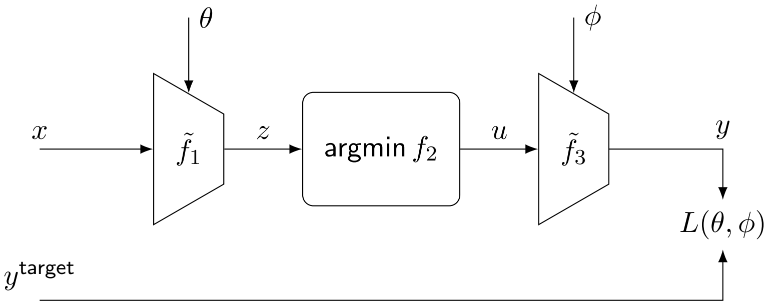
- optimisation problems can be embedded *inside* deep learning models
- back-propagation by either unrolling the optimisation algorithm or implicit differentiation of the optimality conditions
  - the former is easy to implement using automatic differentiation but memory intensive
  - the latter requires that solution be strongly convex locally (i.e., invertible H)
  - but does not need to know how the problem was solved, nor store intermediate forward-pass calculations
  - computing H^{-1} may be costly

Summary and Open Questions

- optimisation problems can be embedded *inside* deep learning models
- back-propagation by either unrolling the optimisation algorithm or implicit differentiation of the optimality conditions
  - the former is easy to implement using automatic differentiation but memory intensive
  - the latter requires that solution be strongly convex locally (i.e., invertible H)
  - but does not need to know how the problem was solved, nor store intermediate forward-pass calculations
  - computing H^{-1} may be costly
- active area of research and many open questions
  - Are declarative nodes slower?
  - Do declarative nodes give theoretical guarantees?
  - How best to handle non-smooth or discrete optimization problems?
  - What about problems with multiple solutions?
  - What if the forward pass solution is suboptimal?
  - Can problems become infeasible during learning?
  - ...

examples and applications

Common Theme



Differentiable Eigen Decomposition

Finding the eigenvector corresponding to the maximum eigenvalue of a real symmetric matrix $X \in \mathbb{R}^{m \times m}$ can be formulated as

$$\begin{array}{ll} \text{maximize (over } u \in \mathbb{R}^m) & u^T X u \\ \text{subject to} & u^T u = 1 \end{array}$$

whose optimality conditions (for solution y) are

$$Xy = \lambda_{\text{max}} y \quad \text{and} \quad y^T y = 1.$$

Differentiable Eigen Decomposition

Finding the eigenvector corresponding to the maximum eigenvalue of a real symmetric matrix $X \in \mathbb{R}^{m \times m}$ can be formulated as

$$\begin{array}{ll} \text{maximize (over } u \in \mathbb{R}^m) & u^T X u \\ \text{subject to} & u^T u = 1 \end{array}$$

whose optimality conditions (for solution y) are

$$Xy = \lambda_{\mathsf{max}} y \quad \text{and} \quad y^T y = 1.$$

Taking derivatives with respect to components of X we get,

$$\frac{\mathrm{d}y}{\mathrm{d}X_{ij}} = -\frac{1}{2}(X - \lambda_{\mathsf{max}} I)^{\dagger}(E_{ij} + E_{ji})y \quad \in \mathbb{R}^m$$

Full Eigen Decomposition

We can extend the previous result to finding **all** eigenvalues of a real symmetric matrix $X \in \mathbb{R}^{m \times m}$,

$$\begin{array}{ll} \text{maximize (over } U \in \mathbb{R}^{m \times m}) & \mathbf{tr}\left(U^T X U\right) \\ \text{subject to} & U^T U = I \end{array}$$

whose optimality conditions (for solution Y) are

$$X y_k = \lambda_k y_k \quad \text{and} \quad Y^T Y = I.$$

Full Eigen Decomposition

We can extend the previous result to finding **all** eigenvalues of a real symmetric matrix $X \in \mathbb{R}^{m \times m}$,

$$\begin{array}{ll} \text{maximize (over } U \in \mathbb{R}^{m \times m}) & \textbf{tr}\left(U^T X U\right) \\ \text{subject to} & U^T U = I \end{array}$$

whose optimality conditions (for solution Y) are

$$X y_k = \lambda_k y_k \quad \text{and} \quad Y^T Y = I.$$

Taking derivatives with respect to components of X we get,

$$\frac{\mathrm{d} y_k}{\mathrm{d} X_{ij}} = -\frac{1}{2}(X - \lambda_k I)^\dagger (E_{ij} + E_{ji}) y_k \quad \in \mathbb{R}^m$$

Eigen Decomposition Backward Pass

Let $Y = [y_1 \ \cdots \ y_m]$, $\Lambda = \textbf{diag}(\lambda_1, \ldots, \lambda_m)$ and $X = Y \Lambda Y^T$. Then with $v_k^T = \frac{\mathrm{d}L}{\mathrm{d}y_k}$,

$$\frac{\mathrm{d}L}{\mathrm{d}X_{ij}} = \sum_{k=1}^{m} v_k^T \frac{\mathrm{d}y_k}{\mathrm{d}X_{ij}}$$

After some algebraic manipulation we can write the gradient with respect to all X as,

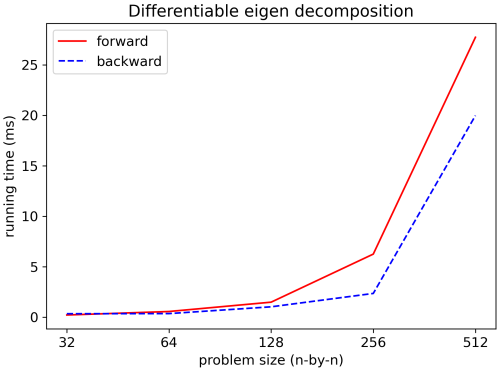
$$\frac{\mathrm{d}L}{\mathrm{d}X} = \frac{1}{2}\left(Y(\tilde{\Lambda} \odot Y^T V)Y^T\right) + \frac{1}{2}\left(Y(\tilde{\Lambda} \odot Y^T V)Y^T\right)^T \quad \in \mathbb{R}^{1 \times m \times m}$$

where $\tilde{\Lambda}_{ij} = \frac{1}{\lambda_i - \lambda_j}$ for $i \neq j$ and zero otherwise.

PyTorch Implementation

```python
class EigenDecompositionFcn(torch.autograd.Function):
    """PyTorch autograd function for eigen decomposition."""

    @staticmethod
    def forward(ctx, X):
        B, M, N = X.shape

        # use torch's eigh function to find the eigenvalues and eigenvectors of a symmetric matrix
        with torch.no_grad():
            lmd, Y = torch.linalg.eigh(0.5 * (X + X.transpose(1, 2)))

            ctx.save_for_backward(lmd, Y)
            return Y

    @staticmethod
    def backward(ctx, dJdY):
        lmd, Y = ctx.saved_tensors
        B, M, N = Y.shape

        # compute all pseudo-inverses simultaneously
        L = lmd.view(B, 1, M) - lmd.view(B, M, 1)
        L = torch.where(torch.abs(L) < eps, 0.0, 1.0 / L)

        # compute full gradient over all eigenvectors
        dJdX = torch.bmm(torch.bmm(Y, L * torch.bmm(Y.transpose(1, 2), dJdY)), Y.transpose(1, 2))
        dJdX = 0.5 * (dJdX + dJdX.transpose(1, 2))

        return dJdX
```

Experiment


Differentiable eigen decomposition
forward
backward
running time (ms)
0
5
10
15
20
25
32
64
128
256
512
problem size (n-by-n)

Optimal Transport

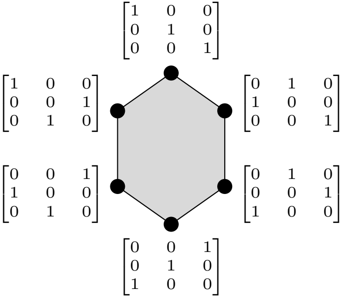
One view of optimal transport is as a matching problem

- from an m-by-n cost matrix M
- to an m-by-n probability matrix P,

often formulated with an entropic regularisation term,

$$\begin{aligned} \text{minimize} \quad & \langle M, P \rangle + \tfrac{1}{\gamma}\langle P, \log P \rangle \\ \text{subject to} \quad & P\mathbf{1} = r \\ & P^T\mathbf{1} = c \end{aligned}$$

with $\mathbf{1}^T r = \mathbf{1}^T c = 1$.



The row and column sum constraints ensure that P is a doubly stochastic matrix (lies within the convex hull of permutation matrices).

Solving Entropic Optimal Transport

Solution takes the form

$$P_{ij} = \alpha_i \beta_j e^{-\gamma M_{ij}}$$

and can be found using the Sinkhorn algorithm,

- Set $K_{ij} = e^{-\gamma M_{ij}}$ and $\alpha, \beta \in \mathbb{R}^n_{++}$
- Iterate until convergence,

$$\alpha \leftarrow r \oslash K\beta$$
$$\beta \leftarrow c \oslash K^T\alpha$$

  where $\oslash$ denotes componentwise division
- Return $P = \mathbf{diag}(\alpha) K \mathbf{diag}(\beta)$

Differentiable Optimal Transport

- Option 1: back-propagate through Sinkhorn algorithm

Differentiable Optimal Transport

- Option 1: back-propagate through Sinkhorn algorithm
- Option 2: use the implicit differentiation result

$$\underbrace{\frac{\mathrm{d}L}{\mathrm{d}M}}_{m\text{-by-}n} = \underbrace{\frac{\mathrm{d}L}{\mathrm{d}P}}_{m\text{-by-}n} \overbrace{\frac{\mathrm{d}P}{\mathrm{d}M}}^{m\text{-by-}n\text{-by-}m\text{-by-}n}$$

Differentiable Optimal Transport

- Option 1: back-propagate through Sinkhorn algorithm
- Option 2: use the implicit differentiation result

$$\underbrace{\frac{\mathrm{d}L}{\mathrm{d}M}}_{\text{1-by-}mn} = \underbrace{\frac{\mathrm{d}L}{\mathrm{d}P}}_{\text{1-by-}mn} \overbrace{\frac{\mathrm{d}P}{\mathrm{d}M}}^{mn\text{-by-}mn}$$

(think of vectorising M and P)

Optimal Transport Gradient

Derivation of the optimal transport gradient is quite tedious (see notes). The result:

$$
\begin{aligned}
\frac{\mathrm{d}L}{\mathrm{d}M} &= \frac{\mathrm{d}L}{\mathrm{d}P}\left(H^{-1}A^T\left(AH^{-1}A^T\right)^{-1}AH^{-1} - H^{-1}\right)B \\
&= \gamma\frac{\mathrm{d}L}{\mathrm{d}P}\textbf{diag}(P)\begin{bmatrix} A_1 \\ A_2 \end{bmatrix}^T \begin{bmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{12}^T & \Lambda_{22} \end{bmatrix} \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \textbf{diag}(P) - \gamma\frac{\mathrm{d}L}{\mathrm{d}P}\textbf{diag}(P)
\end{aligned}
$$

where

$$
\begin{bmatrix} A_1 \\ A_2 \end{bmatrix} = \begin{bmatrix} \mathbf{0}_n^T & \mathbf{1}_n^T & \cdots & \mathbf{0}_n^T \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0}_n^T & \mathbf{0}_n^T & \cdots & \mathbf{1}_n^T \\ I_{n\times n} & I_{n\times n} & \cdots & I_{n\times n} \end{bmatrix}
$$

$$
\begin{aligned}
\left(AH^{-1}A^T\right)^{-1} &= \frac{1}{\gamma}\begin{bmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{12}^T & \Lambda_{22} \end{bmatrix} \\
&= \frac{1}{\gamma}\begin{bmatrix} \textbf{diag}(r_{2:m}) & P_{2:m,1:n} \\ P_{2:m,1:n}^T & \textbf{diag}(c) \end{bmatrix}^{-1}
\end{aligned}
$$

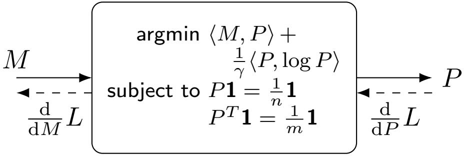
Implementation

```
@staticmethod
def backward(ctx, dJdP):
    # unpacked cached tensors
    M, r, c, P = ctx.saved_tensors
    batches, m, n = P.shape

    # initialize backward gradients (-v^T H^{-1} B)
    dLdM = -1.0 * gamma * P * dLdP

    # compute [vHAt1, vHAt2] = -v^T H^{-1} A^T
    vHAt1, vHAt2 = sum(dJdM[:, 1:m, 0:n], dim=2), sum(dJdM, dim=1)

    # compute [v1, v2] = -v^T H^{-1} A^T (A H^{-1] A^T)^{-1}
    P_over_c = P[:, 1:m, 0:n] / c.view(batches, 1, n)
    lmd_11 = cholesky(diag_embed(r[:, 1:m]) - einsum("bij,bkj->bik", P[:, 1:m, 0:n], P_over_c))
    lmd_12 = cholesky_solve(P_over_c, lmd_11)
    lmd_22 = diag_embed(1.0 / c) + einsum("bji,bjk->bik", lmd_12, P_over_c)

    v1 = cholesky_solve(vHAt1.view(batches, m-1, 1), lmd_11).view(batches, m-1) -
        einsum("bi,bji->bj", vHAt2, lmd_12)
    v2 = einsum("bi,bij->bj", vHAt2, lmd_22) - einsum("bi,bij->bj", vHAt1, lmd_12)

    # compute v^T H^{-1} A^T (A H^{-1] A^T)^{-1} A H^{-1} B - v^T H^{-1} B
    dLdM[:, 1:m, 0:n] -= v1.view(batches, m-1, 1) * P[:, 1:m, 0:n]
    dJdM -= v2.view(batches, 1, n) * P

    # return gradients
    return dJdM
```

Experiment

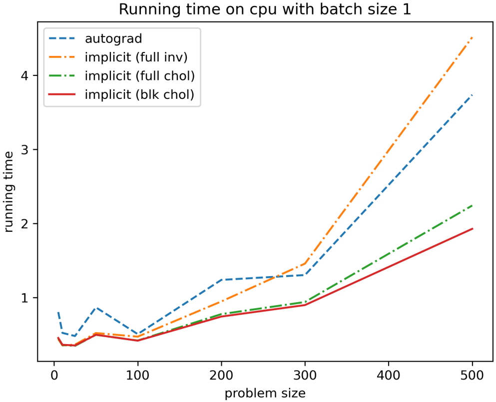
Bi-level optimisation problem with
lower-level optimal transport problem:

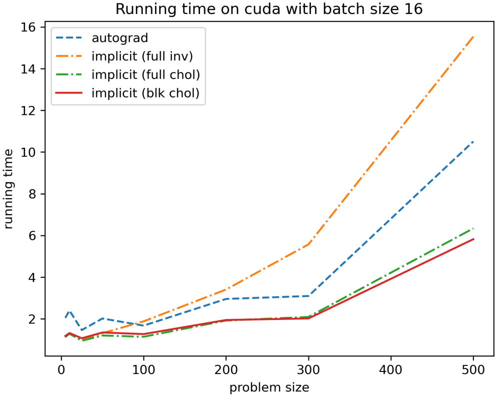
$$
\begin{array}{ll}
\text{minimize} & \frac{1}{2}\|P - P^{\text{target}}\|_F^2 \\
\text{subject to} & \text{minimize } \langle M, P\rangle + \frac{1}{\gamma}\langle P, \log P\rangle \\
& \text{subject to} \quad \begin{array}{l} P\mathbf{1} = \frac{1}{n}\mathbf{1} \\ P^T\mathbf{1} = \frac{1}{m}\mathbf{1} \end{array}
\end{array}
$$



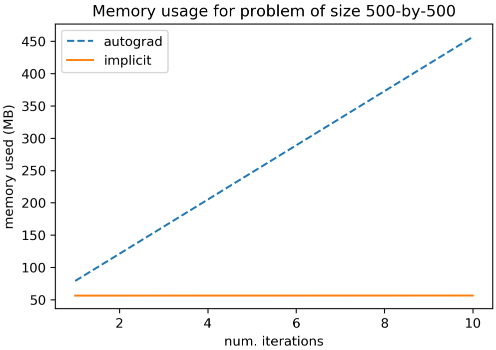
with upper-level variable $M \in \mathbb{R}^{m\times n}$.

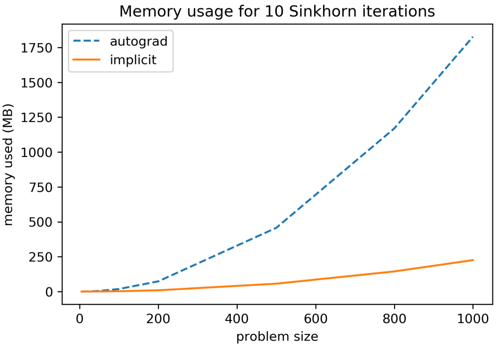
Results: Running Time





Results: Memory Usage





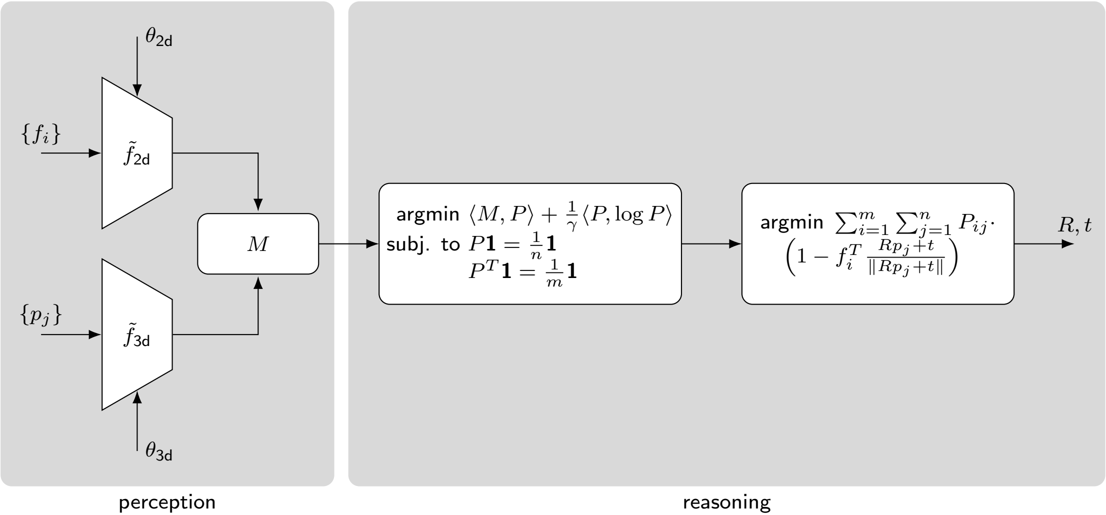
Application to Blind Perspective-n-Point

find the location where the photograph was taken

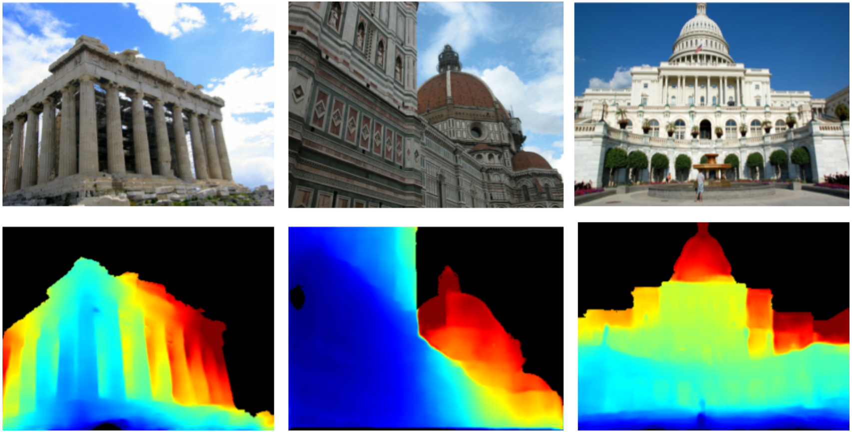
Coupled Problem

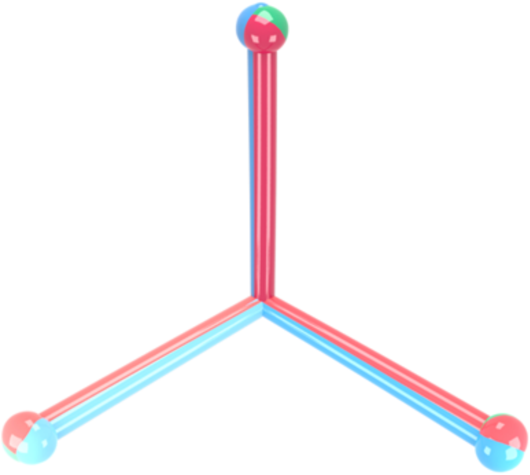
- if we knew **correspondences** then determining **camera pose** would be easy
- if we knew **camera pose** then determining **correspondences** would be easy

Blind Perspective-n-Point Network Architecture

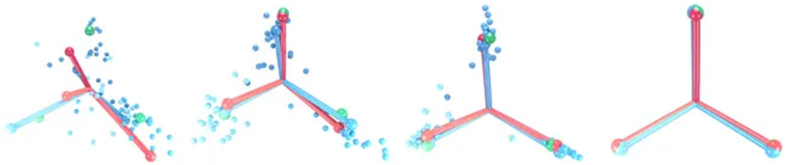


Blind Perspective-n-Point Results





Neural Collapse



- **NC1. Variability collapse.** Features converge to their class means.
- **NC2. Convergence to simplex ETF.** Maximally separated angular classifier vectors.
- **NC3. Convergence to self-duality.** Class means and classifier normal vectors align.
- **NC4. Nearest class-centre.** Classifier selects the class whose mean is closest.

Equiangular Tight Frames (ETFs)

- A **frame** forms an overcomplete basis for a space
- An **equiangular tight frame** has equal norm and equal pairwise inner-product
- A **simplex ETF** has $n = d + 1$ vectors in $\mathbb{R}^d$

$$M = \alpha U \underbrace{\left(I - \frac{1}{n} \mathbf{1}\mathbf{1}^T \right)}_{\text{canonical ETF}}$$

where

- α is arbitrary scale
- U is an arbitrary rotation/reflection

Guiding Neural Collapse

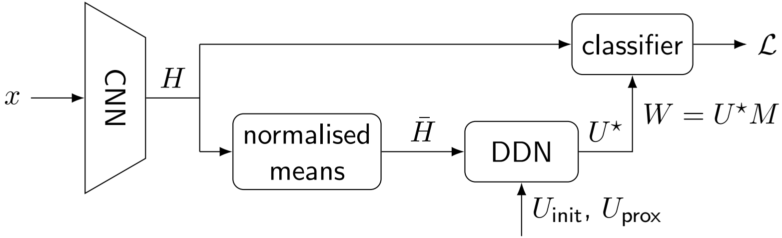
- at each iteration project class means onto the nearest simplex ETF to get the classifier weights by solving

$$\begin{array}{ll} \text{minimize} & \left\|\bar{H} - UM\right\|_F^2 + \frac{\delta}{2}\left\|U - U_{\text{prox}}\right\|_F^2 \\ \text{subject to} & U^T U = I \end{array}$$

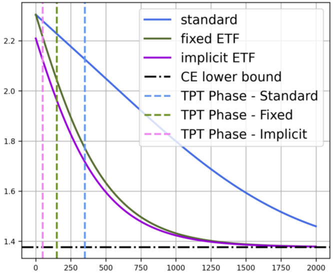
Guiding Neural Collapse

- at each iteration project class means onto the nearest simplex ETF to get the classifier weights by solving

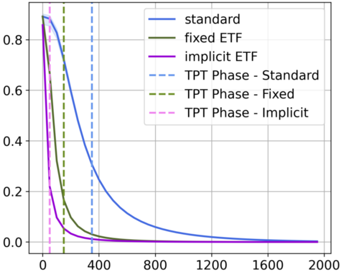
$$\begin{array}{ll} \text{minimize} & \left\|\bar{H} - UM\right\|_F^2 + \frac{\delta}{2}\left\|U - U_{\text{prox}}\right\|_F^2 \\ \text{subject to} & U^T U = I \end{array}$$



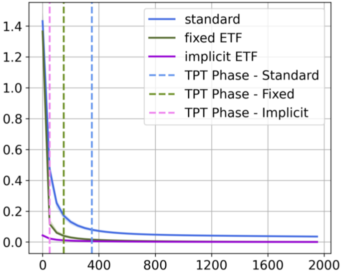
Neural Collapse Results (UFM-10)



cross-entropy loss



NC1. variability collapse



NC3. convergence to self-duality

Further Resources

Where to from here?

- Deep declarative networks (`http://deepdeclarativenetworks.com`)
  - lots of small code examples and tutorials
- CVXPyLayers (`https://github.com/cvxgrp/cvxpylayers`)
- Theseus (`https://sites.google.com/view/theseus-ai`)
- JAXopt (`https://github.com/google/jaxopt`)

lecture notes available at `https://users.cecs.anu.edu.au/~sgould`

end