Lecture Notes on Differentiable Optimisation in Deep Learning

Stephen Gould
stephen.gould@anu.edu.au

May 23, 2024*

Abstract

Optimisation is at the heart of machine learning. In this lecture series, I will formally introduce the notion of an optimisation problem, or mathematical program, and review classic ideas from convex analysis. I will present algorithms for solving optimisation problems and conditions that hold at the optimum (which provides a certificate of optimality). Next, I will show how deep learning can not only be viewed as a large-scale optimisation problem but can include smaller optimisation problems embedded within it, forming so-called bi-level and multi-level optimisation problems. Last, I will explore some practical considerations, open research questions and applications involving differentiable optimisation in deep learning models. The lectures assume a solid understanding of linear algebra and calculus at the first- and second-year undergraduate level[1].

1 Motivation

On 1 January 1801 Italian astronomer Giuseppe Piazzi discovered Ceres, a new planetoid, or dwarf planet, orbiting the sun. Ceres was small and too dim to see by naked eye. Nevertheless, Piazzi managed to track the planetoid using a telescope, making several measurements over 40 days, before further observation became impossible as Ceres became occluded by the sun. Astronomers were keenly interested in observing Ceres when it re-emerged from behind the sun but they lacked the mathematical tools to predict its exact position and trajectory. The problem piqued the interest of Carl Friedrich Gauss, then 24 years old. He developed a model of planetary motion and used the method of **least-squares**, which he had developed a few years earlier, to fit the model to Piazzi's observations [33]. This was perhaps the first application of an optimisation algorithm—the method of least-squares—to scientific discovery. Gauss's predictions were so accurate that it allowed another astronomer, Franz Xaver von Zach, to quickly recover Ceres on 31 December 1801, as it emerged from behind the sun one year after being first discovered.

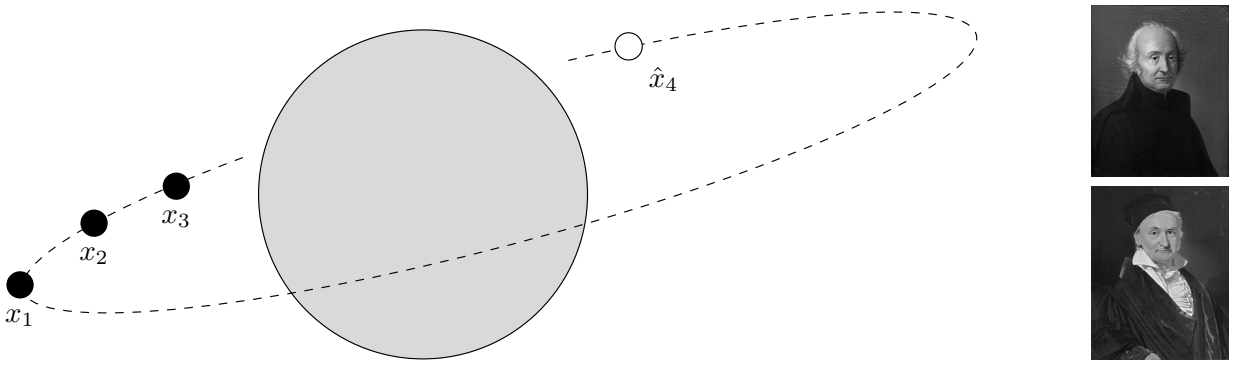


Figure 1: Discovery of Ceres by Piazzi, which led to development of the method of least-squares by Gauss.

Today optimisation is everywhere in science and engineering. In financial mathematics problems involve maximising profits or minimising costs subject to constraints on resources and budgets. In engineering we wish to find the best design amongst a family of candidates, for example, maximising the span of a bridge subject to load constraints or minimising the size of a transistor in a circuit subject to power and timing constraints. In logistics and planning we may wish to find the cheapest way to distribute goods from suppliers to consumers across a network.[2] In robotics for optimal control of dynamical systems. In statistics and data science optimisation is used for curve fitting and data visualisation. And it is at the heart of machine learning and deep learning when we minimise loss functions with respect to the parameters of our model.

*First given at the International School on Artificial Intelligence and its Applications in Computer Science, 15 December 2022.
[1]Textbooks covering the necessary background include Strang [32], Magnus and Neudecker [25], Spivak [31].
[2]A related dual problem is to find the worst bottleneck within a so-called max-flow/min-cut network.

2 Convex Optimisation Review

This lecture covers standard topics in convex optimisation and largely follows the presentation of Boyd and Vandenberghe [6]. Other good reference texts include Bertsekas [4], Nocedal and Wright [27], Hiriart-Urruty and Lemarechal [15], and Rockafellar [29].

2.1 Formal Definition

We can summarise optimisation as

"*finding values for a set of variables that minimises a measure of cost subject to some constraints*"[3]

which is often written in a standard format,

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

or, more formally,

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

Here $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ are the optimisation variables (or decision variables), $f_0 : \mathbb{R}^n \to \mathbb{R}$ is the **objective function** (or cost function or loss), $f_i : \mathbb{R}^n \to \mathbb{R}$ for $i = 1, \ldots, p$ are the **inequality constraint functions**, and $h_i : \mathbb{R}^n \to \mathbb{R}$ for $i = 1, \ldots, q$ are the **equality constraint functions**. If the problem has no explicit constraints ($p = q = 0$) we say that the problem is **unconstrained**. An inequality constraint f_i is **active** at a feasible point x if $f_i(x) = 0$. It is inactive if $f_i(x) < 0$. Denote by

$$\mathcal{D} = \left(\bigcap_{i=0}^{p} \mathbf{dom}\,(f_i) \right) \cap \left(\bigcap_{i=1}^{q} \mathbf{dom}\,(h_i) \right) \subseteq \mathbb{R}^n \tag{2}$$

the **domain** of the optimisation problem. The domain may impose implicit constraints on x. For example, $\log x$ has implicit constraint $x > 0$. Together (the domain and) the inequality and equality constraint functions define the **feasible set**,

$$\left\{ x \in \mathcal{D} \;\middle|\; \begin{array}{ll} f_i(x) \leq 0, & i = 1, \ldots, p \\ h_i(x) = 0, & i = 1, \ldots, q \end{array} \right\}, \tag{3}$$

i.e., the set of vectors x that are in the domain of the problem and satisfy all of the constraints. A **solution**, or optimal point, $x^\star$ has the smallest value of f_0 among all feasible points. A solution may not always exist. The optimal value of the problem is often denoted by $p^\star$ and is equal to $f_0(x^\star)$ when a solution does exists.[4] Note that there may be multiple solutions but the optimal value is unique. Formally, we define the **optimal value** as

$$p^\star = \inf_{x \in \mathcal{D}} \left\{ f_0(x) \;\middle|\; \begin{array}{ll} f_i(x) \leq 0, & i = 1, \ldots, p \\ h_i(x) = 0, & i = 1, \ldots, q \end{array} \right\}. \tag{4}$$

If the problem is **infeasible**, i.e., no x satisfy the constraints, then $p^\star = \infty$. If the problem is unbounded below then $p^\star = -\infty$. We will mostly be interested in problems where $x^\star$ exists and $p^\star$ is finite.

So far we have required $x^\star$ to be a global solution, i.e., have lowest objective value among all feasible x. In some problems it may be too expensive to find the global minimum and we'd be satisfied with a local minima. We say that a point x is **locally optimal** if it is the best solution within some local neighbourhood, formally if there exists an $R > 0$ such that x is optimal for the following problem,

$$\begin{array}{lll} \text{minimize (over } z) & f_0(z) & \\ \text{subject to} & f_i(z) \leq 0 & i = 1, \ldots, p \\ & h_i(z) = 0 & i = 1, \ldots, q \\ & \|z - x\|_2 \leq R. & \end{array} \tag{5}$$

Figure 2 shows example optimal points for one-dimensional unconstrained problems ($n = 1$, $p = q = 0$). An example with two inequality constraints in two dimensions ($n = 2$, $p = 2$, $q = 0$) is shown in Figure 3.

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

[4] While we have tried to be very careful to avoid confusion notation, this is the one place where notation clash is unavailable. The number of equality constraints, denoted by p, and optimal value, denoted by $p^\star$, are very different things.

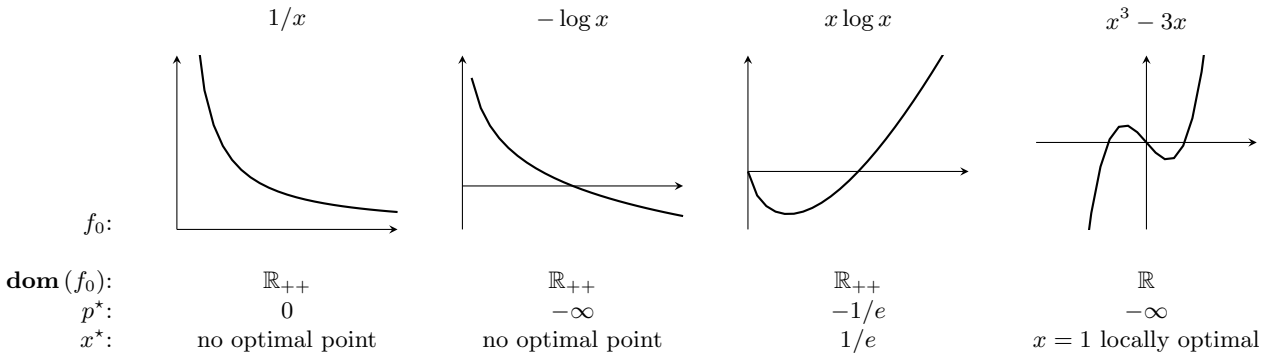


Figure 2: Global and local optimality for one-dimensional (unconstrained) examples.

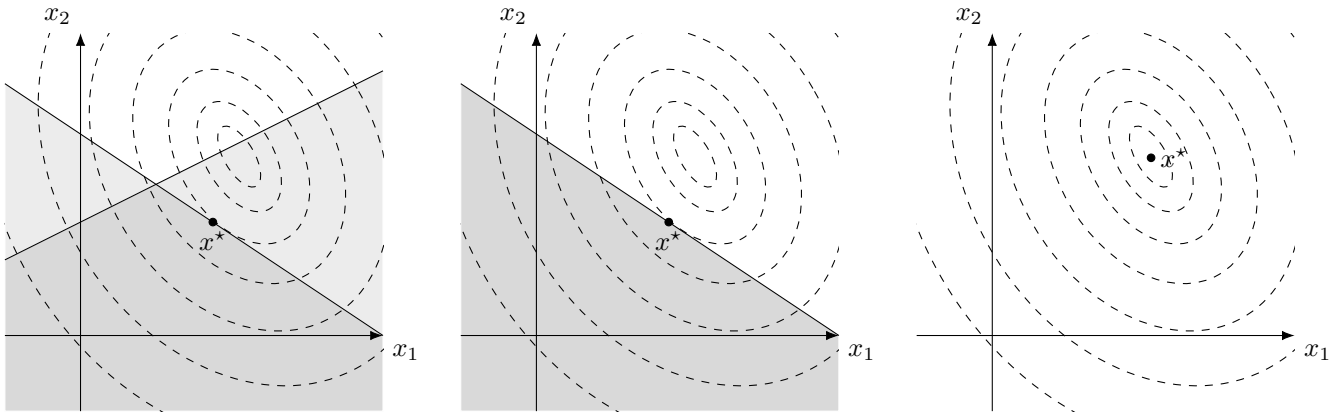


Figure 3: Example two-dimensional constrained optimisation problem with two constraints (one is inactive). Dashed line shows contours of the objective function. Shaded area represents the feasible set. In this example, removing the inactive constraint gives the same solution; while removing the active constraint gives a different solution.

2.2 Least Squares

Let us now revisit the least-squares problem,

$$\text{minimize} \quad \|Ax - b\|_2^2. \tag{6}$$

Assuming that the matrix A^TA is invertible the problem has an analytic solution,

$$x^\star = \left(A^TA\right)^{-1}A^Tb \tag{7}$$

which can be obtained by differentiating the objective and setting it to zero. However, instead of computing the solution directly, efficient iterative algorithms are usually used in practice. The computation time for solving a least-squares problem is $O(n^2m)$ for $A \in \mathbb{R}^{m\times n}$, typically $m \gg n$, and less if A is structured.

The matrix A^TA will not be invertible if A is not full rank. In this case we can still write the solution in closed-form using the singular value decomposition of A. Assume matrix A has rank r and let $A = U\Sigma V^T$. Then Σ is an r-by-r diagonal matrix with non-zero diagonal elements. Expanding the objective function we have,

$$\begin{aligned} f_0(x) \triangleq \|Ax - b\|_2^2 &= (Ax - b)^T(Ax - b) && (8)\\ &= x^TA^TAx - 2b^TAx + b^Tb && (9)\\ &= x^TV\Sigma^2V^Tx - 2b^TU\Sigma V^Tx + b^Tb && (10)\\ &= z^T\Sigma^2z - 2b^TU\Sigma z + b^Tb && (11)\end{aligned}$$

where $z = V^Tx$. Now differentiating with respect to z,

$$\begin{aligned} \nabla f_0(z) &= 2\Sigma^2z - 2\Sigma U^Tb && (12)\\ &= 2\Sigma(\Sigma z - U^Tb) && (13)\\ &= 0 && (14)\end{aligned}$$

Therefore $z^\star = \Sigma^{-1}U^Tb$ and $x^\star = Vz^\star = V\Sigma^{-1}U^Tb$.[5] In fact, any $x^\star + w$ with w in the null space of A is also a valid solution since $A(x^\star + w) = Ax^\star + Aw = Ax^\star$.

Yet another popular method for solving least-squares is via QR factorisation. Let $A = QR$ with $Q^TQ = I$ and R an n-by-n upper triangular matrix. Then

$$\begin{aligned} x^\star &= \left(A^TA\right)^{-1}A^Tb && (15)\\ &= \left(R^TQ^TQR\right)^{-1}R^TQ^Tb && (16)\\ &= \left(R^TR\right)^{-1}R^TQ^Tb && (17)\\ &= R^{-1}R^{-T}R^TQ^Tb && (18)\\ &= R^{-1}Q^Tb && (19)\end{aligned}$$

where multiplication by R^{-1} is implemented via back substitution [32].

Just like Gauss did for estimating the trajectory of Ceres, the most common use of least-squares is fitting a curve to a set of points, or **regression**. Let us consider the example of fitting an n-th order polynomial curve $f_a(x) = \sum_{k=0}^n a_kx^k$ to set of noisy points $\{(x_i, y_i)\}_{i=1}^m$. That is, we assume that the y_i are measured as $y_i = f_a(x_i) + \epsilon_i$ where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ is random Gaussian noise. An example is shown in Figure 4. The $n + 1$ coefficients of the polynomial that best fits the points in terms of minimising the sum-of-squares residual between the points and the curve,

$$\text{argmin}_a \quad \sum_{i=1}^m \left(f_a(x_i) - y_i\right)^2, \tag{20}$$

are found by solving a least-squares optimisation problem,

$$\text{minimize (over } a \in \mathbb{R}^{n+1}) \quad \left\| \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^n \\ 1 & x_2 & x_2^2 & \dots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \dots & x_m^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} - \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \right\|_2^2. \tag{21}$$

Here x and y are data for the problem and a is the optimisation variable (i.e., coefficients of the polynomial that we seek to find).

Least-squares is a special case of convex optimisation, which we will discuss in a little while. However, before going any further we need to introduce some basic ideas from convex analysis.

[5] To obtain $x^\star$ we technically needed to solve $V^Tx = z^\star$ for x. We can see that $x^\star$ is a solution since $V^Tx^\star = V^TVz^\star = Iz^\star = z^\star$.

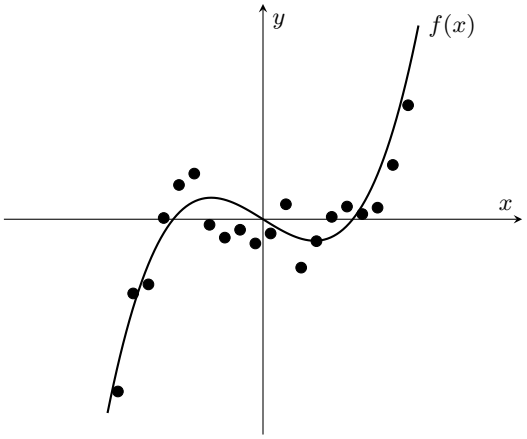


Figure 4: Polynomial curve fitting is a least-squares optimisation problem.

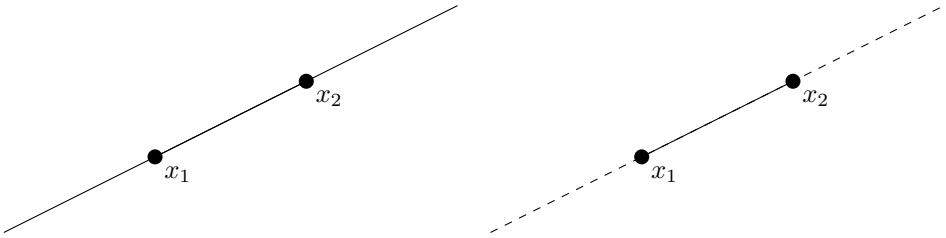


Figure 5: Lines and line segments.

2.3 Convex Sets

A set, in our context, is just a collection of points in $\mathbb{R}^n$. A **line** through any two points x_1 and x_2 defines a set characterised by

$$x = \theta x_1 + (1-\theta)x_2, \quad (\theta \in \mathbb{R}). \tag{22}$$

An **affine set** is a set that contains the line through any two distinct points in the set. For example, the solution to a set of linear equations $\{x \mid Ax = b\}$ defines an affine set. Conversely, every affine set can be expressed as the solution set of a system of linear equations.

A **line segment** between x_1 and x_2 is the set of points

$$x = \theta x_1 + (1-\theta)x_2 \tag{23}$$

with $0 \leq \theta \leq 1$.[6]

A **convex set** is any set $\mathcal{C}$ which contains the line segment between every pair of points in the set,

$$x_1, x_2 \in \mathcal{C} \quad \Longrightarrow \quad \theta x_1 + (1-\theta)x_2 \in \mathcal{C} \text{ for all } 0 \leq \theta \leq 1 \tag{24}$$

This is illustrated in Figure 6. Examples of some typical convex sets are shown in Figure 7. Notice that for all of these sets, the line segment between any two points in the set, lies within the set. **Cones** have the property that arbitrary nonnegative scaling of any point in the cone remains in the cone. Lorentz cones are also convex, but not all cones are convex. The most common convex cones in machine learning are the nonnegative orthant, $\mathbb{R}^n_+$, and the set of positive semindefinite matrices, $\mathbb{S}^n_+$.

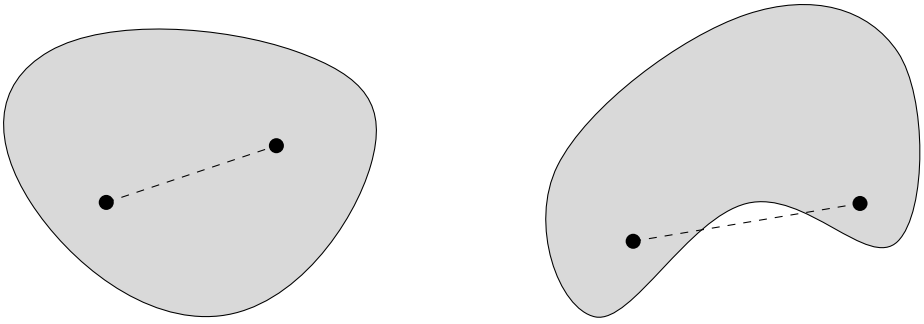
The convex (resp. affine) **hull** $\mathcal{H}$ of an arbitrary set $\mathcal{C}$ is formed by taking the convex (resp. affine) combination of all points in the set. It is the smallest convex (resp. affine) set that contains the set $\mathcal{C}$.

The intersection of an arbitrary number of convex sets is a convex set.

An important result relating to convex sets is the **separating hyperplane theorem**. Roughly speaking, the theorem states that for any two nonempty disjoint convex sets $\mathcal{C}$ and $\mathcal{D}$, there exists a hyperplane with $\mathcal{C}$ on one side and $\mathcal{D}$ on the other. The separation need not be strict (meaning that some points from $\mathcal{C}$ and $\mathcal{D}$ may lie on the hyperplane itself). A consequence of (a variant of) the separating hyperplane theorem is the **supporting hyperplane theorem**, which states that there exists a supporting hyperplane[7] at every boundary point of a convex set.

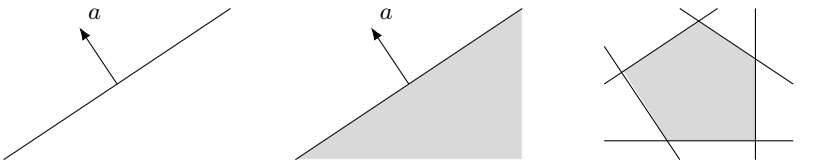
[6] The point $x = \theta x_1 + (1-\theta)x_2$ is called a **convex combination** of x_1 and x_2. This can be extended to an arbitrary number of points, that is $x = \sum_{i=1}^k \theta_i x_i$ with $\sum_{i=1}^k \theta_i = 1$ and $\theta_i \geq 0$ is a convex combination of the points $x_1, \ldots, x_k$.

[7] A supporting hyperplane is one that touches the set at one or more points and has the set fall in the halfspace defined by the hyperplane.



(a) convex (b) nonconvex

Figure 6: Convex and nonconvex sets in 2D. The line segment between any two points in a convex set lies within the set.



(a) Hyperplane, $\{x \mid a^T x = b\}$

(b) Halfspace, $\{x \mid a^T x \leq b\}$

(c) Polyhedron, $\{x \mid Ax \preceq b, Cx = d\}$

(d) Ball, $\{x \mid \|x - x_c\| \leq r\}$

(e) Ellipsoid, $\{Ax + b \mid \|x\|_2 \leq 1\}$

(f) Lorentz Cone, $\{(x, t) \mid \|x\| \leq t\}$

Figure 7: Some common convex sets.

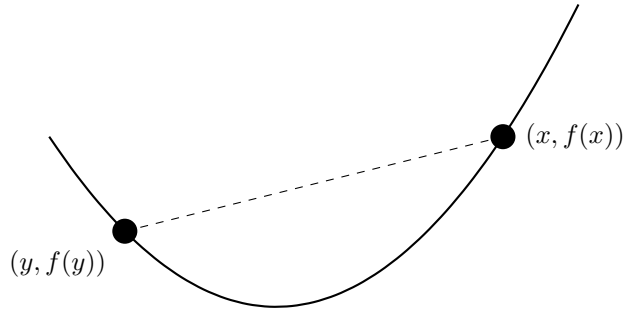


Figure 8: Basic inequality for convex functions.

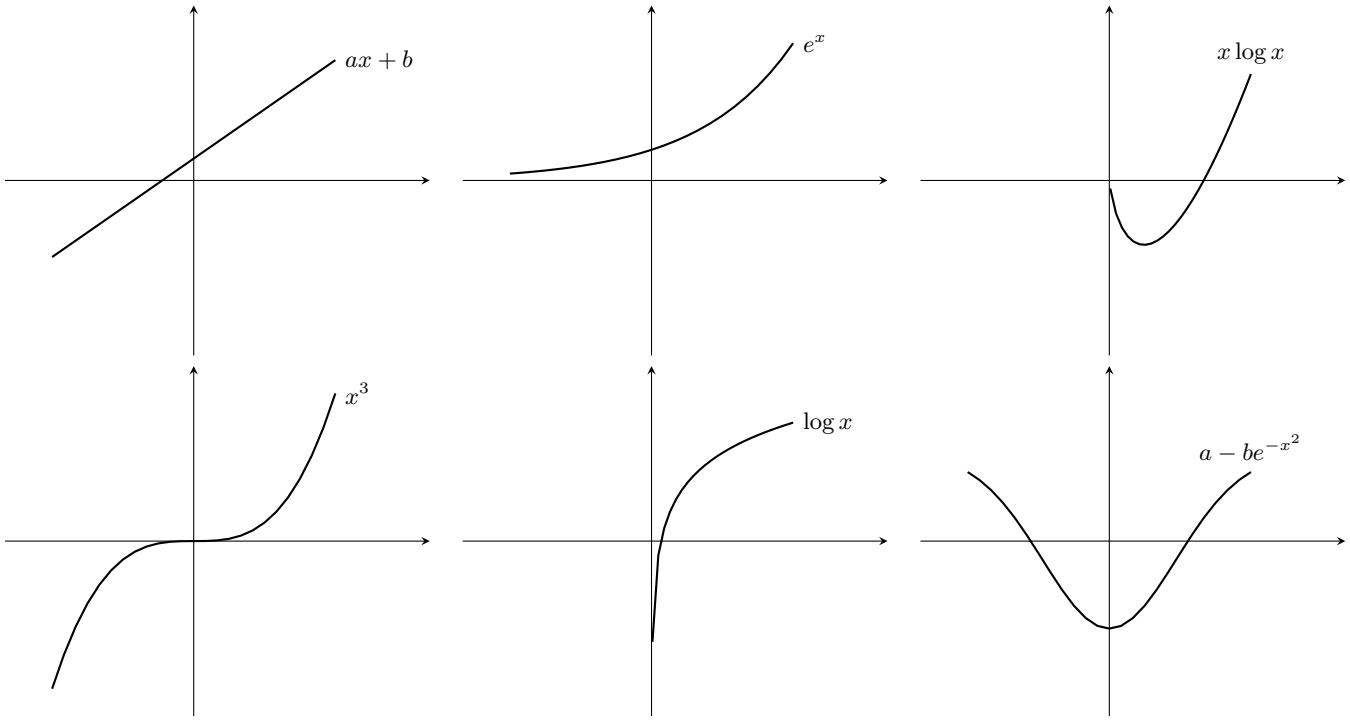


Figure 9: Some example convex functions (top row) and nonconvex functions (bottom row). Linear functions, $ax + b$, are also concave. Of the nonconvex functions (bottom row), only $\log x$ is concave. All other functions are nonconcave.

2.4 Convex Functions

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

$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y) \tag{25}$$

holds for all $x, y \in \mathbf{dom}\,(f), 0 \leq \theta \leq 1$. This is sometimes called Jensen's inequality (which technically extends the inequality to an arbitrary number of points). Graphically this inequality states that the line segment between two points on the function sits above the function as is shown in Figure 8.

Function f is said to be **concave** if its negation $-f$ is convex. Examples of convex and nonconvex functions are shown in Figure 9. Several operations preserve convexity of functions such as taking the nonnegative weighted sum of a set of convex functions, pointwise maximum of a set of functions (i.e., upper envelope), or minimising over a subset or variables. See Figure 10 for two examples.[8]

Convex functions can have straight regions and even a flat bottom. If the inequality in Equation 25 holds strictly (for $0 < \theta < 1$) then the function is said to be strictly convex. Moreover, if the function has minimum positive curvature everywhere (i.e., can be under approximated with a quadratic) then it is called strongly convex. Strong convexity is useful for convergence analysis of optimisation algorithms and guaranteeing a unique minimum. Examples are given in Figure 11.

The **epigraph** of a function is the set formed by all the points above the function.[9] It is useful for linking properties

[8] An interesting observation is that the set of all convex functions forms a convex cone, since it is closed under nonnegative summation.
[9] The hypograph is the set of points lying under a graph.

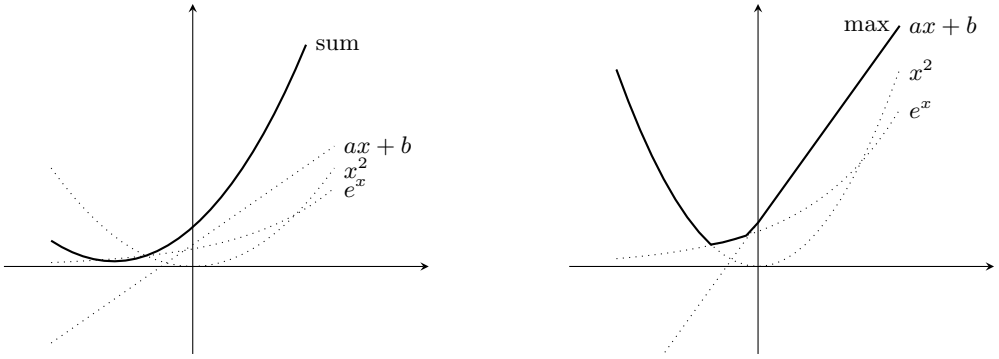


Figure 10: Weighted sum and pointwise maximum (upper envelope) of convex functions is convex.

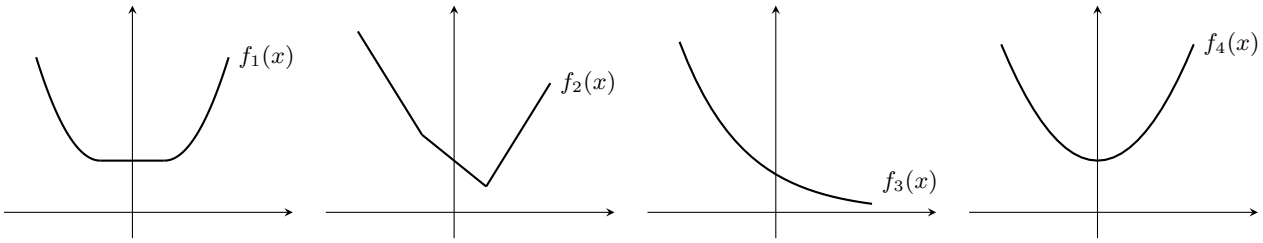


Figure 11: Smooth convex, nonsmooth convex, strictly convex and strongly convex.

we know about convex sets and convex functions. Formally, the epigraph of function $f : \mathbb{R}^n \to \mathbb{R}$ is the set

$$\mathbf{epi}(f) = \{(x,t) \in \mathbb{R}^{n+1} \mid x \in \mathbf{dom}\,(f)\,, f(x) \leq t\}. \tag{26}$$

A function f is convex if and only if its epigraph is a convex set. See Figure 12 for examples.

2.5 Differentiable Convex Functions

A function f is differentiable if $\mathbf{dom}\,(f)$ is open and the gradient

$$\nabla f(x) = \left(\frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \ldots, \frac{\partial f(x)}{\partial x_n}\right) \tag{27}$$

exists at each $x \in \mathbf{dom}\,(f)$. A differentiable function f with convex domain is convex if and only if the following first-order condition is satisfied,

$$f(y) \geq f(x) + \nabla f(x)^T (y - x) \quad \text{for all } x, y \in \mathbf{dom}\,(f)\,. \tag{28}$$

In other words, the first-order approximation of a convex function is a global under estimator. See Figure 13.

A function f is twice differentiable if $\mathbf{dom}\,(f)$ is open and the Hessian $\nabla^2 f(x) \in \mathbb{S}^n$,

$$\nabla^2 f(x)_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j}, \quad i, j = 1, \ldots, n, \tag{29}$$

exists at each $x \in \mathbf{dom}\,(f)$. A twice differentiable function f with convex domain is convex if and only if

$$\nabla^2 f(x) \succeq 0 \quad \text{for all } x \in \mathbf{dom}\,(f)\,. \tag{30}$$

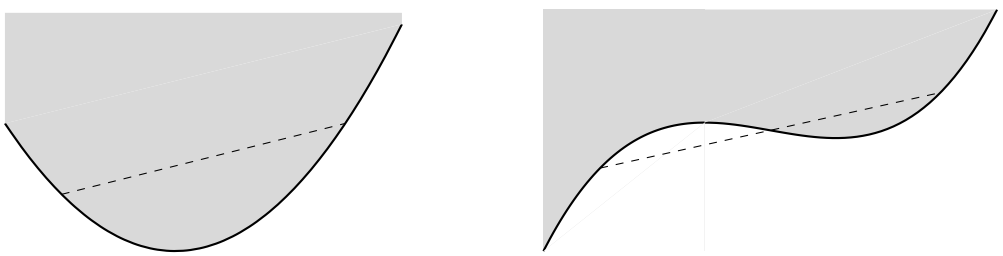


Figure 12: Epigraph of a function, $\mathbf{epi}(f) = \{(x,t) \mid x \in \mathbf{dom}\,(f)\,, f(x) \leq t\} \subseteq \mathbb{R}^{n+1}$. A function f is convex if and only if $\mathbf{epi}(f)$ is a convex set. Example convex function (left) and nonconvex function (right).

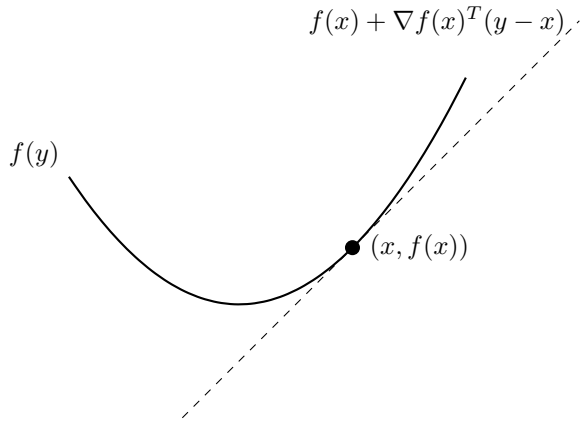


Figure 13: First-order condition for convexity of a differentiable function.

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

Let us end this section by exploring the log-sum-exp function as an important function in machine learning and interesting example of a convex function,

$$f(x) = \log \sum_{k=1}^{n} \exp x_k. \tag{31}$$

This function is twice differentiable so we can test its convexity by examining its Hessian. To do so we write down its partial derivatives with respect to each element of x,

$$\frac{\partial f(x)}{\partial x_i} = \frac{\exp x_i}{\sum_{k=1}^{n} \exp x_k} \tag{32}$$

$$\frac{\partial^2 f(x)}{\partial x_i \partial x_j} = \frac{\left(\sum_{k=1}^{n} \exp x_k\right) [\![i = j]\!] \exp x_i - \exp x_i \exp x_j}{\left(\sum_{k=1}^{n} \exp x_k\right)^2} \tag{33}$$

where the second line comes from differentiating the first line with respect to x_j using the quotient rule. Here $[\![i = j]\!]$ is the indicator function, returning value one if its argument is true and zero otherwise. Readers familiar with machine learning will recognise the first derivative of log-sum-exp as the so-called softmax function. Introducing $z_k = \exp x_k$ we can write the above expression more compactly as

$$\frac{\partial^2 f(x)}{\partial x_i \partial x_j} = \frac{\left(\mathbf{1}^T z\right) [\![i = j]\!] z_i - z_i z_j}{\left(\mathbf{1}^T z\right)^2}. \tag{34}$$

Here $\mathbf{1}^T z$ is shorthand for $\sum_{i=1}^{n} z_i$. The Hessian assembles these second partial derivatives into a matrix,

$$\nabla^2 f(x) = \frac{1}{(\mathbf{1}^T z)^2} \left((\mathbf{1}^T z)\mathbf{diag}(z) - z z^T\right) \tag{35}$$

To show that $\nabla^2 f(x) \succeq 0$, we must verify that $v^T \nabla^2 f(x) v \geq 0$ for all v,

$$v^T \nabla^2 f(x) v = \frac{1}{\left(\mathbf{1}^T z\right)^2} v^T \left((\mathbf{1}^T z)\mathbf{diag}(z) - z z^T\right) v \tag{36}$$

$$= \frac{1}{\left(\mathbf{1}^T z\right)^2} \left((\mathbf{1}^T z) v^T \mathbf{diag}(z) v - (v^T z)^2\right) \tag{37}$$

So showing convexity of the log-sum-exp function amounts to proving that $(\mathbf{1}^T z) v^T \mathbf{diag}(z) v \geq (v^T z)^2$, observing that the multiplicative factor $\frac{1}{(\mathbf{1}^T z)^2}$ is always positive. That is, we need to prove that

$$\left(\sum_{k=1}^{n} z_k\right)\left(\sum_{k=1}^{n} z_k v_k^2\right) \geq \left(\sum_{k=1}^{n} v_k z_k\right)^2, \tag{38}$$

which is a straightforward application of the Cauchy-Schwarz inequality, $(a^T b)^2 \leq \|a\|_2^2 \|b\|_2^2$, with $a = (\sqrt{z_1}, \ldots, \sqrt{z_n})$ and $b = (\sqrt{z_1} v_1, \ldots, \sqrt{z_n} v_n)$. Therefore, log-sum-exp is convex.

A cute alternative proof is to write,

$$v^T \nabla^2 f(x) v = \sum_{k=1}^{n} \theta_k v_k^2 - \left(\sum_{k=1}^{n} \theta_k v_k \right)^2 \tag{39}$$

where $\theta_k = \frac{z_k}{\mathbf{1}^T z}$. Note that since $z_k \geq 0$ we have $\theta_k \geq 0$ and $\mathbf{1}^T \theta = 1$. Thus, the quantity $\sum_{k=1}^{n} \theta_k v_k$ is a convex combination of the v_k. Now, by Jensen's inequality we have

$$g\left(\sum_{k=1}^{n} \theta_k v_k \right) \leq \sum_{k=1}^{n} \theta_k g(v_k) \tag{40}$$

for any convex function g. Letting $g(x) \triangleq x^2$ gives the desired result.

2.6 Convex Optimisation Problems

An optimisation problem,

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, p \\ & h_i(x) = 0, \quad i = 1, \ldots, q \end{array} \tag{41}$$

is convex if the objective function f_0 and all the inequality constraint functions f_i are convex and all the equality constraint functions h_i are affine, $h_i(x) \triangleq a_i^T x - b_i$. The latter is often written in matrix form as $Ax = b$. Under this restriction each constraint defines a convex set, the intersection of which satisfies all constraints and is also a convex set. *So a convex optimisation problem minimises a convex function (equivalently maximises a concave function) over a convex feasible set.*

A key feature of convex optimisation problems is that any locally optimal point is (globally) optimal. This fact can be easily proved.

Proof. Suppose that x is locally optimal, but there exists a feasible y with lower objective, i.e., $f_0(y) < f_0(x)$. Since x is locally optimal there must be an $R > 0$ such that

$$z \text{ feasible and } \|z - x\|_2 \leq R \implies f_0(z) \geq f_0(x)$$

Consider $z = \theta y + (1 - \theta)x$ with $\theta = \frac{R}{2\|y - x\|_2}$. We have that $\|y - x\|_2 > R$ since we assumed $f_0(y) < f_0(x)$, so $0 < \theta < 1/2 < 1$. Therefore z is a convex combination of two feasible points, hence also feasible. Moreover, $\|z - x\|_2 = R/2$ (from our choice of θ) and therefore $f_0(z) \geq f_0(x)$ by our assumption that x is locally optimal. But

$$\begin{aligned} f_0(z) &\leq \theta f_0(y) + (1 - \theta) f_0(x) \\ &< \theta f_0(x) + (1 - \theta) f_0(x) \\ &= f_0(x) \end{aligned}$$

where the first inequality is by the definition of convex function and the second inequality is from our assumption that $f_0(y) < f_0(x)$. We have a contradiction. Therefore every locally optimal point is globally optimal. □

A graphical illustration of the proof is shown in Figure 14.

There are many standard types of convex optimisation problems categorised by the functional form of the objective and constraint functions. Examples include linear programs (LPs), quadratic programs (QPs), quadratically constrained quadratic programs (QCQPs), second-order cone programs (SOCPs) and semidefinite programs (SDPs).

2.7 Optimality Criterion

Without actually solving a given optimisation problem we can state conditions that must hold at any optimal point. Specifically, for differentiable convex objective function f_0, a point x is optimal if and only if it is feasible and $\nabla f_0(x)^T (y - x) \geq 0$ for all feasible points y. If the gradient $\nabla f_0(x)$ is nonzero then this says that $\nabla f_0(x)$ defines a supporting hyperplane for the feasible set $\mathcal{X}$ at point x (see Figure 15). In other words, f_0 cannot be further minimised by moving in a descent direction from x and remaining feasible.

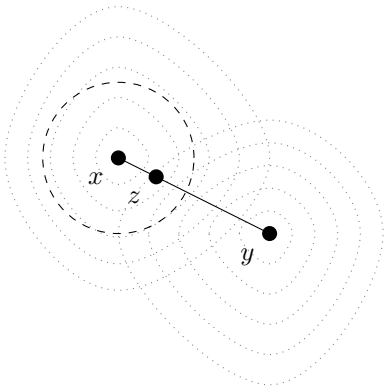


Figure 14: Graphical proof that local optima of convex optimisation problem are global optimal: Towards contradiction, suppose x is locally optimal, but there exists a feasible y with lower objective. Since x is locally optimally there exists a radius R such that no other point within R of x has lower objective (and 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, the objective value at z must be between that at x and y, i.e., lower than $f_0(x)$. We have a contradiction.

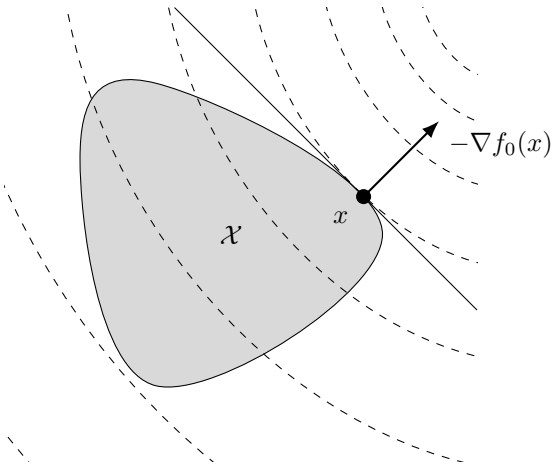


Figure 15: Optimality criterion for differentiable f_0.

2.8 Duality

Duality plays a central role in the theory of convex optimisation. Let us start be defining some auxiliary functions. The **Lagrangian** function $\mathcal{L} : \mathcal{D} \times \mathbb{R}^p \times \mathbb{R}^q \to \mathbb{R}$ for an optimisation problem in standard form (Equation 1), not necessarily convex, is defined as

$$\mathcal{L}(x, \lambda, \nu) = f_0(x) + \sum_{i=1}^{p} \lambda_i f_i(x) + \sum_{i=1}^{q} \nu_i h_i(x). \tag{42}$$

It can be seen as the weighted sum of the objective and constraint functions. Variable λ_i is the Lagrange multiplier associated with the i-th inequality constraint. Likewise, variable ν_i is the Lagrange multiplier associated with the i-th equality constraint. Lagrange multipliers are also called **dual variables** (and x is then the **primal variable**).

From the Lagrangian function we can derive the **dual function**, $g : \mathbb{R}^p \times \mathbb{R}^q$, as

$$g(\lambda, \nu) = \inf_{x \in \mathcal{D}} \mathcal{L}(x, \lambda, \nu) \tag{43}$$

$$= \inf_{x \in \mathcal{D}} \left(f_0(x) + \sum_{i=1}^{p} \lambda_i f_i(x) + \sum_{i=1}^{q} \nu_i h_i(x) \right). \tag{44}$$

The dual function is always concave even if the original problem is nonconvex. This is because it is the pointwise minimum over a set of affine functions (in λ and μ). It can, however, be unbounded below for some values of λ and ν.

An important property of the dual function is that it provides a lower bound for the optimal value of the original optimisation problem for any $\lambda \succeq 0$. That is, if $\lambda \succeq 0$, then $g(\lambda, \nu) \leq p^\star$. The proof for this is straightforward but subtle.

Proof. Let $\tilde{x}$ be any feasible point and $\lambda \succeq 0$. Then

$$f_0(\tilde{x}) \geq \mathcal{L}(\tilde{x}, \lambda, \nu) \geq \inf_{x \in \mathcal{D}} \mathcal{L}(x, \lambda, \nu) = g(\lambda, \nu)$$

where the first inequality comes from observing that for feasibility $h_i(\tilde{x}) = 0$ and $f_i(\tilde{x}) \leq 0$, hence $\sum_{i=1}^{p} \lambda_i f_i(\tilde{x}) \leq 0$. □

It is very natural to try maximise this lower bound, giving rise to the Lagrange dual optimisation problem[10],

$$\begin{array}{ll} \text{maximize} & g(\lambda, \nu) \\ \text{subject to} & \lambda \succeq 0 \end{array} \tag{45}$$

which is (always) a convex optimisation problem with optimal value denoted by $d^\star$. The optimisation variables λ and ν are dual feasible if $\lambda \succeq 0$ and $(\lambda, \nu) \in \mathbf{dom}\,(g)$.

Since g satisfies the lower bound property we have that **weak duality**, $d^\star \leq p^\star$, always holds. This can be used to find nontrivial lower bounds for difficult problems since the dual problem, being convex, may be easier to solve than the original primal problem. A stronger condition, known as **strong duality**, occurs if $d^\star = p^\star$. This does not hold in general but often holds for convex optimisation problems (e.g., LPs and QPs). Tests that guarantee strong duality on convex optimisation problems are called **constraint qualifications** (see Boyd and Vandenberghe [6]).

2.9 KKT Conditions

The following four conditions state what needs to hold at optimality for problems with differentiable objective and constraint functions. They are known as the Karush-Kuhn-Tucker (KKT) conditions and generalise the condition that $\nabla f_0(x) = 0$ for unconstrained problems to constrained problems. The solution to any (differentiable) optimisation problem where strong duality holds must satisfy the KKT conditions. Moreover, for convex optimisation problems, the KKT conditions are sufficient for points to be primal/dual optimal. The conditions on (x, λ, ν) are:

- 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 $i = 1, \ldots, p$

[10] The original problem is known and the **primal problem**.

- the gradient of the Lagrangian with respect to x vanishes,

$$\nabla f_0(x) + \sum_{i=1}^{p} \lambda_i \nabla f_i(x) + \sum_{i=1}^{q} \nu_i \nabla h_i(x) = 0 \tag{46}$$

The last condition states that negative gradient of the objective $-\nabla f_0(x)$ lies within the union of the conic hull of the gradients of the inequality constraint functions $\nabla f_i(x)$ and span of the gradients of the equality constraint functions $\nabla h_i(x)$, sometimes called the **normal cone**. Mathematically,

$$-\nabla f_0(x) \in \mathbf{cone}(\nabla f_i(x) \mid i = 1, \ldots, p) \cup \mathbf{span}(\nabla h_i(x) \mid i = 1, \ldots, q) \tag{47}$$

which is essentially the same as the optimality criterion discussed in Section 2.7.

2.10 Algorithms

There are a very great number of algorithms for solving convex and nonconvex optimisation problems. We will only scratch the surface here and present the most vanilla variants and in all cases assume that our problems are convex and that objective and constraint functions twice continuously differentiable. Let us start with the most straightforward case of unconstrained optimisation,

$$\text{minimize} \quad f_0(x). \tag{48}$$

2.10.1 Gradient Descent

Gradient descent is a simple iterative algorithm that proceeds from an initial starting point x in the domain of the objective function and keeps taking steps in the negative gradient direction to reduce the value of the objective until some stopping criterion is met, typically when the gradient norm is below some threshold or a maximum number of iterations is exhausted. In summary,

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

2.10.2 Line Search

Let Δx denote the search or step direction. For gradient descent this is just $-\nabla f_0(x)$ but there are many other possibilities. For example, coordinate descent chooses as Δx one of the canonical directions e_k, and later we will see other mechanisms for choosing the search direction. An important consideration for any descent algorithm is how big a step to take in the search direction (Line 2 above). Too large a step and we could overshoot the minimum (or risk leaving the domain of f_0). Too small a step and we are wasting compute, slowing down convergence. Here there are three standard strategies. First, we could decide ahead of time on a step size schedule, e.g., by setting t to a small constant or starting with some initial t and decaying with each iteration. Second, we can perform an exact line search to find the minimum value of the objective along the search direction,

$$t^\star = \text{argmin}_{t>0}\, f_0(x + t\Delta x). \tag{49}$$

Last, we can perform an approximate line search using a backtracking procedure to trade-off taking a big step with making sufficient progress on decreasing the objective value. Standard backtracking line search has two parameters: $\alpha \in (0, \frac{1}{2})$ that controls the above mentioned trade-off and $\beta \in (0, 1)$ that controls the granularity of the search. Starting with $t = 1$, backtracking line search repeatedly reduces t to βt until the following condition is satisfied,

$$f_0(x + t\Delta x) < f_0(x) + \alpha t \nabla f_0(x)^T \Delta x. \tag{50}$$

The procedure is illustrated in Figure 16. Conceptually there is a step size t_0 that occurs when the line with damped gradient $f_0(x) + \alpha t \nabla f_0(x)^T \Delta x$ intersects the function $f_0(x + t\Delta x)$. Backtracking line search stops at the first $t < t_0$.

2.10.3 Newton's Method

Even with exact line search the gradient descent algorithm can be slow to converge as the example in Figure 17 shows. One way to speed convergence is to choose a search direction that takes into account the curvature of the objective function. Netwon's method does this by choosing the search direction as

$$\Delta x_{\text{nt}} = -\nabla^2 f_0(x)^{-1} \nabla f_0(x) \tag{51}$$

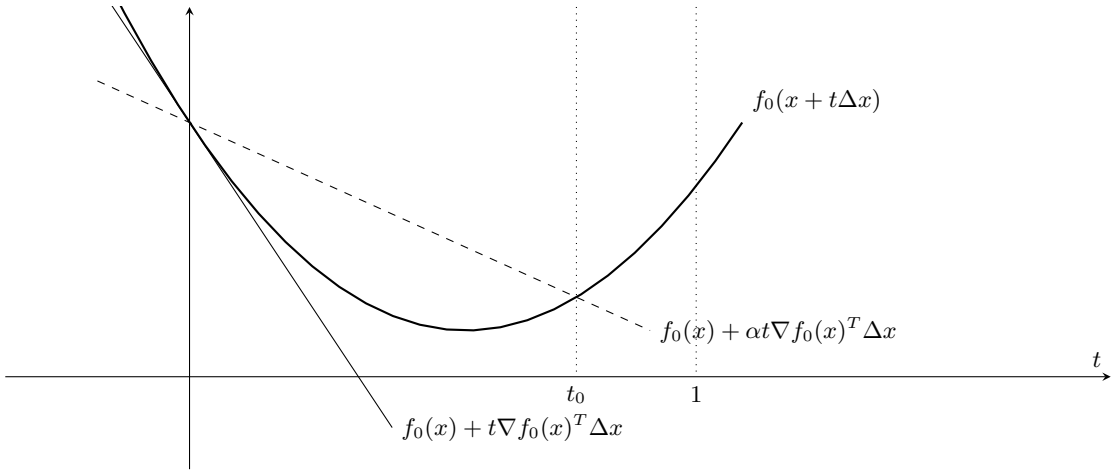


Figure 16: Backtracking line search.

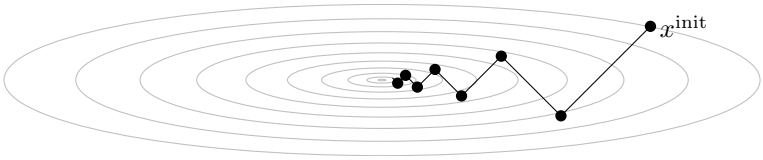


Figure 17: Example gradient descent on function $f_0(x) = x_1^2 + \gamma x_2^2$ with $\gamma \gg 1$.

which can be thought of as being the value of v that minimises the second-order approximation of f_0 at x,

$$\hat{f}(x+v) = f_0(x) + \nabla f_0(x)^T v + \frac{1}{2} v^T \nabla^2 f_0(x) v. \tag{52}$$

Newton's method proceeds in a similar fashion to gradient descent with Δx_{nt} taking the place of the negative gradient $-\nabla f_0(x)$. It is much faster to converge than gradient descent at the expense of having to calculate the inverse Hessian. Indeed, for the problem shown in Figure 17 Newton's method converges in one iteration! For small to medium size problems (of up to a few thousand variables) the added expense is worth it.[11]

2.10.4 Equality Constrained Optimisation

We now turn our attention to equality constrained problems,

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & Ax = b \end{array} \tag{53}$$

where $A \in \mathbb{R}^{q \times n}$ with $\mathbf{rank}(A) = q$ (and $b \in \mathbf{range}(A)$ else the problem is infeasible). For such equality constrained problems we know that $x^\star$ is optimal if and only if there exists a $\nu^\star$ such that

$$\nabla f_0(x^\star) + A^T \nu^\star = 0 \quad \text{and} \quad Ax^\star = b. \tag{54}$$

Starting at a feasible point x a modified Newton step Δx_{nt} can be found by solving

$$\begin{bmatrix} \nabla^2 f_0(x) & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} v \\ w \end{bmatrix} = \begin{bmatrix} -\nabla f_0(x) \\ 0 \end{bmatrix} \tag{55}$$

for variable v. The system of equations can be viewed as solving the optimality conditions for a quadratic approximation of the constrained optimisation problem,

$$\begin{array}{ll} \text{minimize (over } v) & \hat{f}(x+v) \triangleq f_0(x) + \nabla f_0(x)^T v + \frac{1}{2} v^T \nabla^2 f_0(x) v \\ \text{subject to} & A(x+v) = b \end{array} \tag{56}$$

Observe that the second row in Equation 55 ensures that the x iterates stay feasible, since $Av = 0$ and therefore for feasible x we have $A(x + t\Delta x_{\text{nt}}) = Ax + tA\Delta x_{\text{nt}} = Ax = b$.

[11] For larger problems quasi-Newton methods, such as L-BFGS [19], are very popular and offer a trade-off between second-order methods and first-order gradient descent. When we get to deep learning with millions of parameters even L-BFGS becomes too expensive and we are forced to use first-order methods.

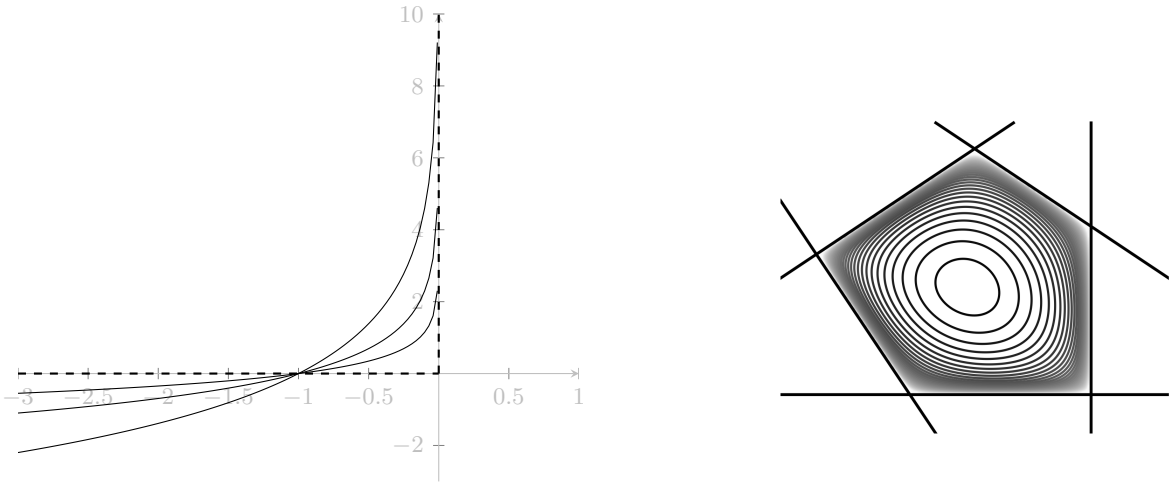


Figure 18: Left: Log-barrier approximation to the indicator function, $-\frac{1}{t}\log(-u)$. Right: contour plot for log-barrier of a two-dimensional polyhedron, $\phi(x) = -\frac{1}{t}\sum_{i=1}^{p}\log(b_i - a_i^T x)$. The log-barrier function climbs very sharply as we approach the boundary of the feasible set from the inside. It has infinite value outside of the feasible set.

2.10.5 Inequality Constrained Optimisation

The classic approach to dealing with inequality constrained optimisation problems,

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, p \\ & Ax = b \end{array} \tag{57}$$

is to first reformulate the problem by moving the inequality constraint functions into the objective via composition with an indicator function,

$$\begin{array}{ll} \text{minimize} & f_0(x) + \sum_{i=1}^{p} I_{\mathbb{R}_-}(f_i(x)) \\ \text{subject to} & Ax = b \end{array} \tag{58}$$

where $I_{\mathbb{R}_-}(u) = 0$ if $u \leq 0$ and $I_{\mathbb{R}_-}(u) = \infty$ otherwise. The reformulation is exactly equivalent to the original problem. We then approximate the indicator with a logarithmic barrier function, which is twice differentiable. This results in an equality constrained problem parametrized by variable t that closely approximates the original problem,

$$\begin{array}{ll} \text{minimize} & f_0(x) - \frac{1}{t}\sum_{i=1}^{p}\log(-f_i(x)) \\ \text{subject to} & Ax = b. \end{array} \tag{59}$$

The larger the value of t the closer the logarithmic barrier approximates the indicator function (see Figure 18). We typically start with a small t and solve the resulting equality constrained problem using Netwon's method. We then repeatedly increase t and, starting with the solution in hand, solve the new equality constrained problem giving better and better approximations to the original problem.

2.10.6 Large Scale Optimisation

For very large scale problems, e.g., as occur in deep learning, Newton's method is too expensive. Even computing the true gradient may be too expensive. Fortunately it is typical for machine learning loss functions (i.e., unconstrained objectives) to decompose over the training data $\{(x_i, y_i)\}_{i=1}^{m}$,

$$L(\theta) = \frac{1}{m}\sum_{i=1}^{m}\ell(f(x_i;\theta), y_i) \tag{60}$$

A method that works well for such problems is to approximate the gradient on a subset (or mini-batch) of the training data $\mathcal{I} \subseteq \{1, \ldots, m\}$ to give,

$$\widehat{\nabla_\theta L} = \frac{1}{|\mathcal{I}|}\sum_{i\in\mathcal{I}}\nabla_\theta\ell(f(x_i;\theta), y_i) \tag{61}$$

The so-called stochastic gradient descent (SGD) algorithm uses this approximation instead of the true gradient in a gradient descent procedure with decaying step size.[12] Under mild assumptions the expected value of the estimated

[12]Here a line search is not used because evaluation of the objective function, i.e., loss function is so expensive and would be needed at multiple locations along the line search.

gradient equals the true gradient, $E\left[\widehat{\nabla_\theta L}\right] = \nabla_\theta L$, and it can be shown that SGD converges to the optimal solution of convex problems [28]. In practice, we randomly permute $[m]$ (to avoid statistical biases in how the training data is ordered) and iterate through adjacent fixed-length intervals to determine $\mathcal{I}$. One pass through the data is called an epoch. Variants of SGD such as AdamW [23] are the most popular methods used in deep learning. There are many, many other methods tailored for all sorts of optimisation problems.

2.11 Summary

Convex optimisation and analysis is an incredibly rich field of study with a fascinating history. This brief introduction has necessarily just touched the surface. There are many topics that we have omitted—generalised inequalities, problems involving matrix arguments, conjugate functions, Frenchel duality, non-smooth optimisation [26], convergence analysis, numerical linear algebra, etc. The interested reader is encouraged to dig deeper by reading the standard texts cited at the beginning of this lecture.

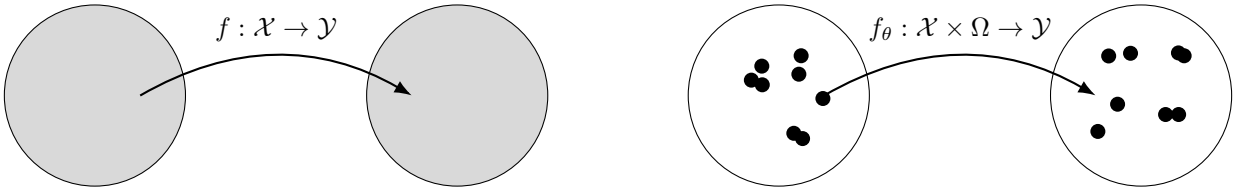


Figure 19: Machine learning from 10,000ft.

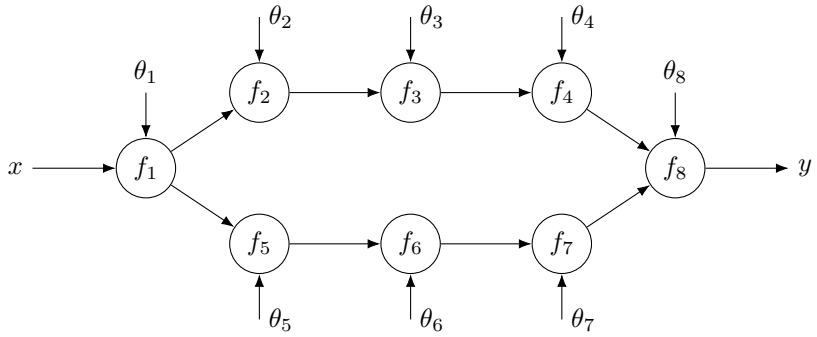


Figure 20: Example of a deep learning model as an end-to-end computation graph. This graph implements the composed function $y = f_8(f_4(f_3(f_2(f_1(x)))), f_7(f_6(f_5(f_1(x)))))$ where each f_i's parameters have been omitted for brevity. Note here that f_8 takes three arguments: one the output from f_4, another the output from f_7, and the last parameters θ_8.

3 Differentiable Optimisation and Deep Learning

This lecture is based on material from Gould et al. [12] and Dontchev and Rockafellar [10]. Other good references include Dempe and Franke [9], Gould et al. [11] and Amos [1].

One view of machine learning (especially supervised classification and regression tasks) is to find a function f that maps from an input space $\mathcal{X}$ to an output space $\mathcal{Y}$ (see Figure 19). Since we can't practically search over all possible mappings we define a function class parametrized by θ and train a model on samples from $\mathcal{X}$ and $\mathcal{Y}$ to minimise (over θ) some loss function L, which typically decomposes over sampled input-output pairs,

$$\text{minimize} \quad \textstyle\sum_{(x,y)\sim\mathcal{X}\times\mathcal{Y}} L(f_\theta(x), y). \tag{62}$$

This is called **empirical risk minimisation** [35]. Here the loss function L tells us **what** to do, and the parametrized function f_θ tells us **how** to do it. The objective (composed of the loss L and the mapping function f) is, in general, nonconvex in the parameters θ. As such, we can only hope to find a local minima of the learning problem.[13]

In deep learning the function f_θ is a composition of simple differentiable parametrized sub-functions, and the parameters are optimised end-to-end. The composed function can be represented by a computation graph as illustrated in Figure 20, where the sub-functions are depicted as nodes in the graph. This type of representation is very popular for describing deep learning architectures where nodes can denote anything from a very simple arithmetic operation to very complicated algorithmic procedures and data transformations.

To compute the derivative of a loss function at the output of the graph, with respect to any parameter or input of the graph, we simply apply the chain rule of differentiation by following the arrows backwards through the graph. This is known as **back propagation**. Two examples are shown in Figure 21 for computing the derivative of the loss L with respect to parameter θ_7 and parameter θ_1, respectively. Here writing out the chain rule we have

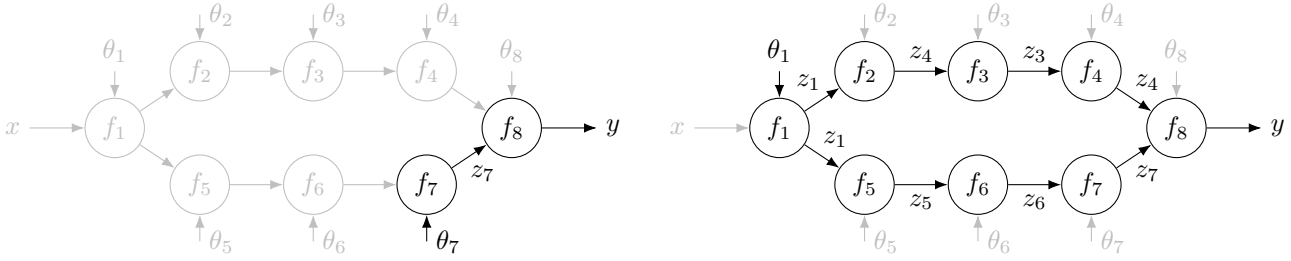
$$\frac{\partial L}{\partial \theta_7} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial z_7}\frac{\partial z_7}{\partial \theta_7} \qquad \text{and} \qquad \frac{\partial L}{\partial \theta_1} = \frac{\partial L}{\partial y}\left(\frac{\partial y}{\partial z_4}\frac{\partial z_4}{\partial z_3}\frac{\partial z_3}{\partial z_2}\frac{\partial z_2}{\partial z_1} + \frac{\partial y}{\partial z_7}\frac{\partial z_7}{\partial z_6}\frac{\partial z_6}{\partial z_5}\frac{\partial z_5}{\partial z_4}\right)\frac{\partial z_1}{\partial \theta_1} \tag{63}$$

where, for the latter derivative $\frac{\partial L}{\partial \theta_1}$, the first term in summation is from the top branch and second term in summation is from the bottom branch of the graph.

So a deep learning node has two distinct operations. In the **forward pass** the node computes the output y as a function of its input x and model parameters θ.[14] In the **backward pass**, which is used during training, the node

[13] In practical applications suboptimal solutions are often desirable since finding the globally optimal solution can result in poor generalization of the model to unseen test samples. Techniques such as regularisation, data-augmentation, and model selection on a hold-out validation set all help in this regard, but are beyond the scope of these lectures.

[14] Not all nodes will have parameters and sometimes parameters are shared between different nodes. The former case is trivial. The latter is also easily handled by summing over paths in the backward pass but we won't consider it further in these lectures.



(a) Computing $\frac{\partial L}{\partial \theta_7}$.

(b) Computing $\frac{\partial L}{\partial \theta_1}$.

Figure 21: Back-propagation of gradients through the computation graph. See text for details.

must compute the derivative of the loss L with respect to the input x (and model parameters θ) given the derivative of the loss with respect to the output y. The fact that gradients can be propagated throughout the whole network makes the deep learning model **end-to-end learnable**.

3.1 Notation

Before proceeding it is worth clarifying notation. For scalar-valued functions $f : \mathbb{R} \to \mathbb{R}$ we denote by

$$\frac{\mathrm{d}f}{\mathrm{d}x} \tag{64}$$

the total derivative of function f with respect to argument x. If a function takes more than one argument we can differentiate with respect to each argument separately by taking partial derivatives denoted by, for example,

$$\frac{\partial f(x,y)}{\partial x}. \tag{65}$$

What is often confusing is when y is also a function of x. Then calculus dictate that the total derivative with respect to x is the sum of direct and indirect terms,

$$\frac{\mathrm{d}f(x,y)}{\mathrm{d}x} = \frac{\partial f(x,y)}{\partial x} + \frac{\partial f(x,y)}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}x}. \tag{66}$$

We already saw in the previous lecture that for multi-dimensional functions $f : \mathbb{R}^n \to \mathbb{R}$ we denote the gradient by

$$\nabla f(x) = \left(\frac{\mathrm{d}f}{\mathrm{d}x_1}, \ldots, \frac{\mathrm{d}f}{\mathrm{d}x_n}\right) \tag{67}$$

which is an n-dimensional (column) vector. More generally, for multi-dimensional vector-valued functions, $f : \mathbb{R}^n \to \mathbb{R}^m$ we define

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = \begin{bmatrix} \frac{\mathrm{d}f_1}{\mathrm{d}x_1} & \cdots & \frac{\mathrm{d}f_1}{\mathrm{d}x_n} \\ \vdots & \ddots & \vdots \\ \frac{\mathrm{d}f_m}{\mathrm{d}x_1} & \cdots & \frac{\mathrm{d}f_m}{\mathrm{d}x_n} \end{bmatrix} \tag{68}$$

as the m-by-n matrix of total derivatives. Note that this is the transpose of ∇f for scalar-valued functions. For functions with signature $f : \mathbb{R}^n \times \mathbb{R}^{\tilde{n}} \to \mathbb{R}^m$ we can also define the matrix of partial derivatives,

$$\frac{\partial}{\partial x} f(x,y) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}. \tag{69}$$

This convention makes the chain rule particularly easy to express by, for example, replacing the scalar products in Equation 66 with matrix multiplications and following the same ordering of expressions. We also have that the affine function $y = Ax + b$ has derivative $\frac{\mathrm{d}y}{\mathrm{d}x} = A \in \mathbb{R}^{m \times n}$ without having to introduce transposes, which is nice.[15]

Symbols D and D_X are also used to denote the derivative operators, which is cleaner that writing $\frac{\mathrm{d}}{\mathrm{d}x}$ and $\frac{\partial}{\partial x}$, especially with inline text, but is less familiar to students so won't be used here. Some authors use D_1 for partial derivatives with respect to the first argument, D_2 for the second, etc.

Last, and perhaps most importantly, authors (including ourselves) are sometimes sloppy with notation and the reader should carefully check the intent from the context.

[15] One drawback of this convention is that the gradients propagated through deep learning networks are with respect to a scalar-valued loss function L and, in all frameworks, stored transposed so that they have the same dimensionality/shape as the variable with respect to which the derivative is taken.

3.2 A Linear Example

To make the above concrete, consider a node that computes a linear function of its input,

$$y = Ax. \tag{70}$$

This could be, for example, a convolution layer in a convolutional neural network (CNN) or a linear layer in a multi-layer perceptron (MLP). Here $x \in \mathbb{R}^n$ serves as the input and $A \in \mathbb{R}^{m \times n}$ are the parameters. (We neglect the bias term for brevity). Back-propagating the derivative of the loss through the node we have

$$\frac{\mathrm{d}L}{\mathrm{d}x} = \frac{\mathrm{d}L}{\mathrm{d}y} A \tag{71}$$

where $\frac{\mathrm{d}L}{\mathrm{d}x} \in \mathbb{R}^{1 \times n}$ is the outgoing derivative of the loss with respect to the input and $\frac{\mathrm{d}L}{\mathrm{d}y} \in \mathbb{R}^{1 \times m}$ is the incoming derivative of the loss with respect to the output. Remember gradients are propagated backwards. Observe that the computational costs of the forward pass and backward pass are identical, $O(mn)$ in this case, and less if A is structured.

We can also compute the derivative of the loss with respect to any or all of the parameters. Consider a single parameter A_{ij}. We have

$$\frac{\mathrm{d}L}{\mathrm{d}A_{ij}} = \frac{\mathrm{d}L}{\mathrm{d}y} \frac{\mathrm{d}y}{\mathrm{d}A_{ij}} = \frac{\mathrm{d}L}{\mathrm{d}y} E_{ij} x = \frac{\mathrm{d}L}{\mathrm{d}y_i} x_j \tag{72}$$

where we obtained $E_{ij}x$ from $\frac{\mathrm{d}y}{\mathrm{d}A_{ij}} = \frac{\mathrm{d}Ax}{\mathrm{d}A_{ij}} = \frac{\mathrm{d}A}{\mathrm{d}A_{ij}} x + A \frac{\mathrm{d}x}{\mathrm{d}A_{ij}}$.

As a prelude to later material, we can also consider the case where we move A to the left-hand side. That is, finding the output y that is the solution to a system of linear equations parametrized by A and x,

$$Ay = x. \tag{73}$$

Here we assume that $A \in \mathbb{R}^{n \times n}$ is full rank so that a unique solution, $y = A^{-1}x$, always exists. Unlike the previous example, we now need to solve a system of equations in the forward pass, which is a lot more work, costing $O(n^3)$ in general, and less if A has some structure (e.g., diagonal, triangular, orthonormal, etc.). Notwithstanding this additional computational cost in obtaining y from x (and A), we can still compute derivatives of the loss with respect to both x and A. Following the same approach as above, we have

$$\frac{\mathrm{d}L}{\mathrm{d}x} = \frac{\mathrm{d}L}{\mathrm{d}y} A^{-1} \tag{74}$$

and

$$\frac{\mathrm{d}L}{\mathrm{d}A_{ij}} = -\frac{\mathrm{d}L}{\mathrm{d}x_i} y_j \tag{75}$$

where we have used the identity $\frac{\mathrm{d}A^{-1}}{\mathrm{d}t} = -A^{-1} \frac{\mathrm{d}A}{\mathrm{d}t} A^{-1}$ and substituted for already computed quantities ($\frac{\mathrm{d}L}{\mathrm{d}x}$ and y) to simplify where appropriate. Letting $v^T = \frac{\mathrm{d}L}{\mathrm{d}y}$ and $w^T = \frac{\mathrm{d}L}{\mathrm{d}x}$ we can obtain w by solving the adjoint system $v = A^T w$. Importantly, any factorization of A used to solve for y in the forward pass can be reused in the backwards pass. This makes the backward pass much faster than the forward pass for any A with nontrivial structure.

3.3 Automatic Differentiation

Automatic differentiation (AD) is an algorithmic procedure that produces code for computing exact[16] derivatives of functions implemented in software code. This is different from numeric gradient approximation by, say, finite differences. It assumes that calculations are composed of a small set of elementary operations that we know how to differentiate. This includes basic arithmetic, exponentiation, logarithms, and trigonometric functions. Automatic differentiation is the workhorse of modern machine learning that greatly reduces development effort.

There are two main flavours of automatic differentiation:

- **forward mode** fixes an independent variable u and computes derivatives $\frac{\mathrm{d}v}{\mathrm{d}u}$ for all dependent variables v
- **reverse mode** fixes a dependent variable v and computes derivatives $\frac{\mathrm{d}v}{\mathrm{d}u}$ for all independent variables u

[16] Up to machine precision.

procedure 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

procedure 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

Figure 22: Toy example for demonstrating automatic differentiation. The forward pass code implements the famous Babylonian algorithm for computing $y = \sqrt{x}$. The backward pass code is automatically generated by unrolling the Babylonian algorithm to compute $\mathrm{d}L/\mathrm{d}x = (\mathrm{d}L/\mathrm{d}y)(\mathrm{d}y/\mathrm{d}x)$ given $\mathrm{d}L/\mathrm{d}y$. Of course, this example is only for illustration: knowing that the forward pass returns $\sqrt{x}$ we could simply hand code the backward pass as $\mathrm{d}L/\mathrm{d}x = (\mathrm{d}L/\mathrm{d}y)(1/(2y))$ using the fact that $\mathrm{d}y/\mathrm{d}x = 1/(2y)$.

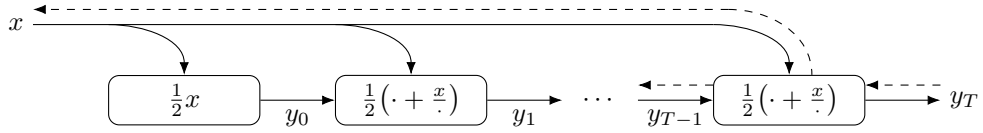


Figure 23: Computation graph for the Babylonian algorithm from Figure 22. The forward and backward passes are denoted by solid and dashed arrows, respectively.

The difference between forward and reverse mode automatic differentiation boils down to the order in which terms in the expression for the gradient are evaluated. As a concrete example, consider again the derivative of the loss L with respect to parameter θ_7 shown in Figure 21. Forward mode automatic differentiation evaluates this expression from right to left as,

$$\frac{\partial L}{\partial \theta_7} = \frac{\partial L}{\partial y}\underbrace{\left(\frac{\partial y}{\partial z_7}\frac{\partial z_7}{\partial \theta_7}\right)}_{\partial y/\partial \theta_7} \tag{76}$$

so by the end we have computed $\partial/\partial\theta_7$ for all variables z_7, y, and L.

Reverse mode automatic differentiation evaluates the other way around, i.e., from left to right,

$$\frac{\partial L}{\partial \theta_7} = \underbrace{\left(\frac{\partial L}{\partial y}\frac{\partial y}{\partial z_7}\right)}_{\partial L/\partial z_7}\frac{\partial z_7}{\partial \theta_7} \tag{77}$$

so by the end we have computed $\partial L/\partial y$, $\partial L/\partial z_7$, and $\partial L/\partial \theta_7$.

The same pattern extends to longer chains and multiple differentiation paths such as for computing $\partial L/\partial\theta_1$ in Equation 63. Due to computational advantages of reverse mode automatic differentiation for computing gradients over multiple parameters (and scalar loss function), it is preferred for deep learning in a process known as back-propagation, which also re-uses intermediate calculations where possible.

Different deep learning frameworks use slightly different approaches (explicit graph construction versus eager evaluation and operator tracking). But in all frameworks the developer only needs to implement the forward pass operation for a node and the backward pass is automatically generated. In general, for each line of the forward pass code, `P, Q = foo(A, B, C)`, automatic differentiation needs to produce a line `dLdA, dLdB, dLdC = foo_vjp(dLdP, dLdQ)` in the backward pass code. Here `vjp` stands for "vector-Jacobian product" and is a term used in deep learning to denote the matrix operations that implement the chain rule in reverse mode. Usually the inputs `A`, `B` and `C`, and outputs `P` and `Q` are also made available to `foo_vjp` through a so-called context variable. We'll see some actual examples in Section 4.

A toy example of automatic differentiation is shown in Figure 22. Here the forward function implements an algorithm known as the Babylonian algorithm for computing the square-root of its argument.[17] Each step in the Babylonian

[17] It is an interesting exercise to analyse the convergence of this algorithm. Hint: first establish that $\sqrt{x}$ is a fixed point. Then show that $y_t^2 \geq x$ for $t > 1$ and, together with $y_t \geq 0$, hence $y_t - y_{t-1} \leq 0$ so that y_t is a decreasing sequence bounded below by $\sqrt{x}$.

```
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;
}
```

Figure 24: Fast inverse square root implementation from Quake III Arena. (Source: Wikipedia)

algorithm is differentiable so automatic differentiation can generate the backward pass code by unrolling the forward pass iterations using the chain rule of differentiation,

$$\frac{\mathrm{d}y_t}{\mathrm{d}x} = \frac{\partial y_t}{\partial x} + \frac{\partial y_t}{\partial y_{t-1}} \frac{\mathrm{d}y_{t-1}}{\mathrm{d}x} \tag{78}$$

$$= \frac{1}{2y_{t-1}} + \frac{1}{2}\left(1 - \frac{x}{y_{t-1}^2}\right) \frac{\mathrm{d}y_{t-1}}{\mathrm{d}x}. \tag{79}$$

The corresponding computation graph for this process is depicted in Figure 23. Note that the intermediate values y_t computed in the forward pass are re-used in the backward pass calculation. Now, in this toy example we could simply have implemented the backward pass using

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}\sqrt{x}}{\mathrm{d}x} = \frac{1}{2\sqrt{x}} = \frac{1}{2y} \tag{80}$$

since we know that the forward pass computes $y = \sqrt{x}$.[18] However, the beauty of automatic differentiation is that for significantly more complicated operations we do not need to figure out the gradient manually. We now turn our attention to a more interesting example of forward pass code that computes the *inverse* of the square-root.

While being a wonderfully powerful tool that has revolutionised machine learning, automatic differentiation does not always work. Specifically if the implementation of the forward pass function contains steps which are not differentiable (even if the overall mathematical function itself is differentiable) then we cannot use automatic differentiation. How might this happen? Well, consider the C code shown in Figure 24, which is the fast inverse square root implementation from Quake III Arena (from the early 1990s). This operation is needed when normalizing vectors, which is a common operation in 3D graphics and therefore must be made to run very fast. The code (which is not actually C-standard compliant) makes use of bit manipulations on the IEEE floating-point number representations to get a rough approximation to the inverse square-root and then performs a single Newton step update. This already gives a highly accurate result (and the second Newton step is commented out). At the time, running on a CPU, the code was about four times faster than computing and inverting the standard library function `sqrt(x)`.

Because of the bit manipulations the Quake III Arena code cannot be automatically differentiated. But the mathematical expression $y = 1/\sqrt{x}$ is clearly differentiable, namely,

$$\frac{\mathrm{d}y}{\mathrm{d}x} = -\frac{1}{2}\frac{1}{x^{3/2}} = -\frac{1}{2}y^3. \tag{81}$$

So in some situations we may want to optimise the forward pass code to make it run faster but the resulting code may then be non-differentiable *even if the mathematical function that we are implementing is differentiable*. In other cases we might just be able to write a faster implementation of the backward pass code than can be generated automatically. For these reasons deep learning frameworks allow developers to implement their own backward pass code. If not implemented then, by default, automatic differentiation is used.

[18] We can even see this by re-examining the Babylonian algorithm, which we know converges to $\sqrt{x}$. Indeed, its easy to see that $y_t = \sqrt{x}$ is a fixed point of the iterates $y_t \leftarrow \frac{1}{2}\left(y_{t-1} + \frac{x}{y_{t-1}}\right)$. Thus, $\frac{\partial y_t}{\partial y_{t-1}} = 0$ as $t \to \infty$ (while $\frac{\mathrm{d}y_t}{\mathrm{d}y_{t-1}} = 1$) and we have $\frac{\mathrm{d}y_t}{\mathrm{d}x} \to \frac{1}{2y_t}$.

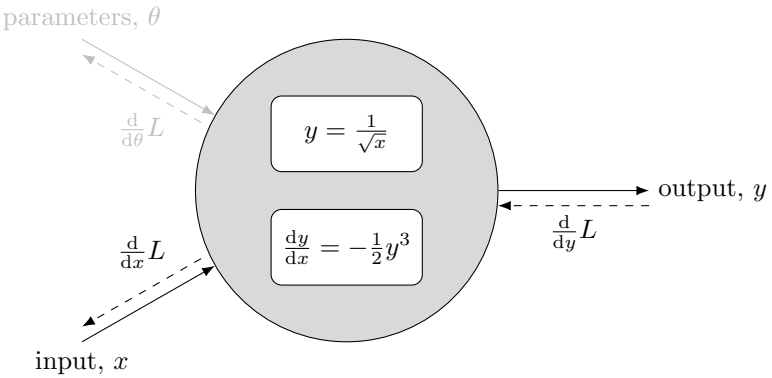


Figure 25: A deep learning node for computing $\frac{1}{\sqrt{x}}$. If using the fast code from Quake III Arena (Figure 24) then automatic differentiation cannot be used and the developer must provide an implementation of the backward pass.

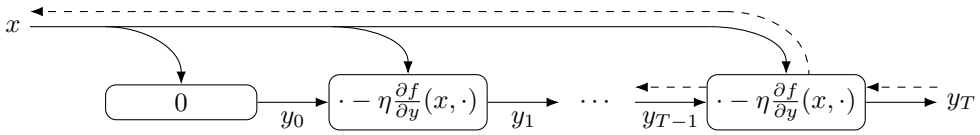


Figure 26: Computation graph for gradient descent in unconstrained optimisation.

The same unrolling approach that we used for the Babylonian algorithm can be applied to gradient descent. Figure 26 shows the computation graph for gradient descent to find the minimum y of some smooth function f conditioned on input x. That is, $y \in \arg\min_u f(x, u)$. In this case we initialise y to some arbitrary value, say zero, and iteratively update its value by taking steps in the negative gradient direction,

$$y_t \leftarrow y_{t-1} - \eta\frac{\partial f}{\partial y}(x, y_{t-1}). \tag{82}$$

In the backward pass we can compute $\frac{\mathrm{d}}{\mathrm{d}x}L$ through recursive evaluation of

$$\frac{\mathrm{d}y_t}{\mathrm{d}x} = \frac{\partial y_t}{\partial x} + \frac{\partial y_t}{\partial y_{t-1}}\frac{\mathrm{d}y_{t-1}}{\mathrm{d}x} \tag{83}$$

$$= -\eta\frac{\partial^2 f}{\partial x \partial y}(x, y_{t-1}) + \left(I - \eta\frac{\partial^2 f}{\partial y^2}(x, y_{t-1})\right)\frac{\mathrm{d}y_{t-1}}{\mathrm{d}x} \tag{84}$$

using back-propagation. However, instead of back-propagating through the optimisation iterates, consider what happens at convergence, i.e., when $y_t = y_{t-1}$ in Equation 84. Dropping subscripts t, we have,

$$\frac{\mathrm{d}y}{\mathrm{d}x} = -\eta\frac{\partial^2 f}{\partial x \partial y}(x, y) + \left(I - \eta\frac{\partial^2 f}{\partial y^2}(x, y)\right)\frac{\mathrm{d}y}{\mathrm{d}x} \tag{85}$$

which after rearranging gives

$$\eta\frac{\partial^2 f}{\partial y^2}(x, y)\frac{\mathrm{d}y}{\mathrm{d}x} = -\eta\frac{\partial^2 f}{\partial x \partial y}(x, y) \tag{86}$$

$$\therefore\ \frac{\mathrm{d}y}{\mathrm{d}x} = -\left(\frac{\partial^2 f}{\partial y^2}(x, y)\right)^{-1}\frac{\partial^2 f}{\partial x \partial y}(x, y) \tag{87}$$

Here we have effectively decoupled calculation of the forward and backward passes since it doesn't really matter how we found the solution y, we can compute its gradient directly using only derivatives of the function f (and knowing the solution y). Importantly, intermediate calculations from the forward pass, i.e., the iterates y_t, do not need to be cached for use during the backward pass, only the converged solution y. We will make this notion, as well as the result for calculating the gradient $\frac{\mathrm{d}y}{\mathrm{d}x}$, more formal and more general below.

3.4 Imperative versus Declarative Nodes

Let us now introduce the notion of a **declarative node**. Everything we've discussed up till now has been what I will call an **imperative node**[19] where the relationship between input x and output y is defined explicitly, i.e., $y = \tilde{f}(x)$

[19]The nomenclature "imperative" and "declarative" is borrowed from the programming languages community.

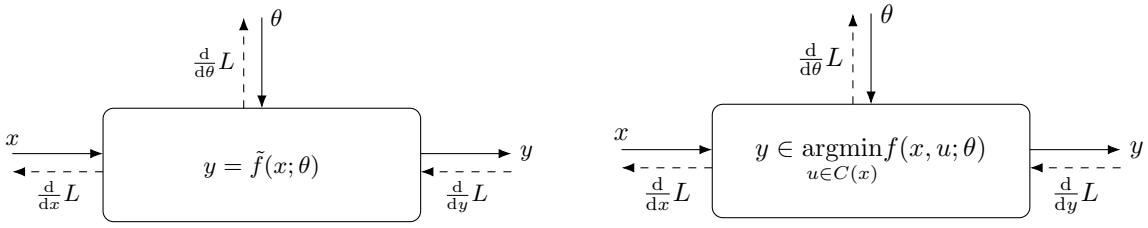


Figure 27: Parametrized data processing nodes in an end-to-end learnable model with global objective or loss function L. During the forward evaluation pass of an imperative node (left) the input x is transformed into output y based on some explicit parametrized function $\tilde{f}(\cdot;\theta)$. During the forward evaluation pass of a declarative node (right) the output y is computed as the minimizer of some parametrized objective function $f(x,\cdot;\theta)$. During the backward parameter update pass for either node type, the gradient of the global objective function with respect to the output $\frac{\mathrm{d}}{\mathrm{d}y}L$ is propagated backwards via the chain rule to produce gradients with respect to the input $\frac{\mathrm{d}}{\mathrm{d}x}L$ and parameters $\frac{\mathrm{d}}{\mathrm{d}\theta}L$. (Reproduced from Gould et al. [12]).

for some (differentiable) function $\tilde{f}$. By contrast, in a declarative node, the input-output relationship is specified implicitly as the solution to an optimisation problem[20],

$$y \in \underset{u \in C(x)}{\operatorname{argmin}} f(x,u;\theta). \tag{88}$$

Both imperative and declarative nodes can be parametrized and during training of the deep learning model these parameters are updated. The difference between the two types of nodes is summarised in Figure 27. Importantly, imperative and declarative nodes can co-exists in the same computation graph.

To make the distinction between imperative and declarative nodes clear, it is worthwhile to consider a concrete example: global average pooling. Here we are given a set of vectors $\{x_i \in \mathbb{R}^m \mid i = 1,\dots,n\}$ and want to compute their mean y. This is easily expressed imperatively as,

$$y = \frac{1}{n}\sum_{i=1}^{n} x_i. \tag{89}$$

Alternatively, we can express the operation declaratively,

$$y = \operatorname{argmin}_{u \in \mathbb{R}^m} \sum_{i=1}^{n} \|u - x_i\|^2 \tag{90}$$

which can be thought of as finding the vector u that minimises the average squared-distance to each of the vectors x_i, i.e., the mean.[21] This doesn't immediately appear like progress. However, the change in perspective opens up new possibilities, such as robust global average pooling, by replacing the ℓ_2 penalty on distances to some arbitrary penalty function ϕ,

$$y = \operatorname{argmin}_{u \in \mathbb{R}^m} \sum_{i=1}^{n} \phi(u - x_i). \tag{91}$$

This is not so easily done with imperative nodes.

The above example shows that declarative nodes subsume imperative nodes in the sense that every imperative node can be rewritten as a declarative node, but not vice versa. This is not to suggest that writing every node as a declarative node is practically useful.

3.5 Bi-level Optimisation: Stackelberg Games

Before proceeding further with discussions on declarative nodes and differentiable optimisation, let us first introduce the idea of bi-level optimisation or so-called Stackelberg games [37]. Consider two players, one **leader** and one **follower** competing in an economic market. The market dictates the price that they are willing to pay for some goods based on the total supply. That is, if the leader and follower produce q_1 and q_2 amount of goods, respectively, then consumers will pay $P(q_1 + q_2)$ dollars per unit of good, where P is some price function.

[20] For this reason some authors refer to declarative nodes as implicit layers.

[21] It is interesting to note that this is a least-squares problem "minimise $\|Au - b\|^2$" where A is constructed by stacking m-by-m identity matrices n times, and b constructed by stacking the x_i.

Each player has a cost structure associated with producing goods, say $C_i(q_i)$, and wants to maximise their profits, $q_i P(q_1 + q_2) - C_i(q_i)$. In a Stackelberg game, the leader picks a quantity of goods to produce first knowing that the follower will respond in an optimal way. In other words, the leader must solve the **bi-level** optimisation problem,

$$\begin{array}{ll} \text{maximize (over } q_1) & q_1 P(q_1 + q_2) - C_1(q_1) \\ \text{subject to} & q_2 \in \operatorname{argmax}_q \, qP(q_1 + q) - C_2(q). \end{array} \tag{92}$$

The leader's problem is known as the **upper-level problem** and the follower's problem (which the leader must also solve) is known as the **lower-level problem**. We can write bi-level optimisation problems in a more familiar way for machine learning with an upper-level loss function and general lower-level optimisation problem as,

$$\begin{array}{ll} \text{minimize (over } x) & L(x, y) \\ \text{subject to} & y \in \operatorname{argmin}_{u \in C(x)} \, f(x, u). \end{array} \tag{93}$$

There exist three common strategies for solving bi-level optimisation problems [2]. First, if a closed-form solution exists for y in the lower-level problem, then we can substitute for y in the upper-level problem and attempt to solve the resulting single level optimisation problem directly using standard means.

Second, for convex lower-level problems we can replace lower-level problem with sufficient conditions for optimality (e.g., the KKT conditions), and solve the equivalent constrained optimisation problem jointly over x and y,

$$\begin{array}{ll} \text{minimize (over } x, y) & L(x, y) \\ \text{subject to} & h(x, y) = 0. \end{array} \tag{94}$$

The downside of this approach is that the constraints many be very difficult to deal with, especially for large scale problems such as occur in deep learning.

Last, we could attempt a gradient descent optimisation approach by computing the gradient of the lower-level solution y with respect to x and use the chain rule of differentiation to get the total gradient of the loss L with respect to x,

$$x \leftarrow x - \eta \left(\frac{\partial L(x, y)}{\partial x} + \frac{\partial L(x, y)}{\partial y} \frac{\mathrm{d}y}{\mathrm{d}x} \right) \tag{95}$$

This requires differentiating the `argmin` operator in the lower-level problem, which may be done by either back-propagating through the optimisation procedure used to solve it or via implicit differentiation as we will see.[22]

3.6 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 **parametrized optimization problems** indexed by continuous variable $x \in \mathbb{R}^n$,

$$\left\{ \begin{array}{lll} \text{minimize (over } u \in \mathbb{R}^m) & f_0(x, u) & \\ \text{subject to} & f_i(x, u) \leq 0, & i = 1, \ldots, p \\ & h_i(x, u) = 0, & i = 1, \ldots, q \end{array} \right\}_{x \in \mathbb{R}^n} \tag{96}$$

that are embedded within the deep learning computation graph. This extends the idea of bi-level optimisation from a composition of two optimisation problems to a graph containing of an arbitrary number of optimisation problems and processing functions that glue them together.

For convenience we will think of inputs (the parameters) and outputs (the solutions) to parametrized optimisation problems as vectors. In many applications they will be more elaborate data structures, e.g., matrices and tensors. The results presented in this lecture readily extend to such data structures, albeit with some additional care in implementation and housekeeping.

A pictorial example of a parametrized optimisation problem that helps to explain the concept is shown in Figure 28. As can be seen changing x results in a new optimisation problem, which when minimised gives a value y for that x. In other words, we can view y as a function of x, sometimes referred to as the **best-response function** in the literature. The main question that we need to answer for gradient descent based learning is:

$$\textit{How do we compute } \tfrac{d}{dx} \, \textit{arg}\,\textit{min}_u \, f(x, u)\textit{?}$$

To answer that question we turn to Dini's implicit function theorem [17], which gives a tool for computing gradients between variables when one is not an explicit function of the other.

[22] Some other techniques for solving bi-level optimisation problems include value function and evolutionary methods. However, these are less amenable to embedding optimisation problems into deep learning architectures and are beyond the scope of these lectures.

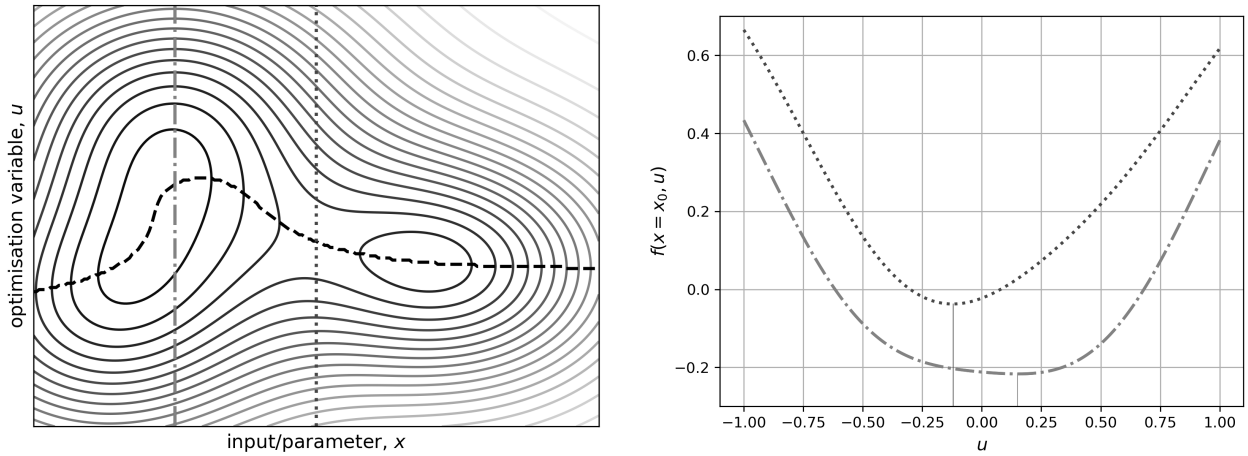


Figure 28: Illustration of a parametrized optimisation problem. The left panel shows contours for a two-dimensional function $f(x, u)$. For each value of x we define an optimisation problem $y = \arg\min_u f(x, u)$. Two example objective functions of u for fixed x are shown in the right panel. Solving for all values of x traces out the so-called best-response function $y(x)$ denoted by the dashed line in the left panel. This may be a set-valued function in general, i.e., more than one y for a given x.

3.7 Dini's Implicit Function Theorem

The following is adapted from Dontchev and Rockafellar [10, p19] where we consider the solution mapping associated with the equation $f(x, u) = 0$,

$$Y : x \mapsto \{u \in \mathbb{R}^m \mid f(x, u) = 0\} \text{ for } x \in \mathbb{R}^n \tag{97}$$

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

Theorem 3.1: (Dini Classic Implicit Function Theorem [10]). Let $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$ be continuously differentiable in a neighbourhood of (x, u) and such that $f(x, u) = 0$, and let $\frac{\partial}{\partial u} f(x, u)$ be nonsingular. Then the solution mapping Y has a single-valued localization y around x for u which is continuously differentiable in a neighbourhood $\mathcal{X}$ of x with Jacobian satisfying

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

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

Roughly speaking, Dini's implicit function theorem tells us that around solutions (x, y) to the implicit equation $f(x, y) = 0$ we can define a local function $y(x)$ that describes how variable y changes as a function of x. Moreover, the theorem gives us the derivative of that function.

Let us illustrate the implicit function theorem by considering the trivial example of differentiating the equation of the unit circle,

$$x^2 + y^2 = 1 \tag{98}$$

depicted graphically in Figure 29. For every value of $x \in (-1, 1)$ there are two distinct solutions for y, namely, $\pm\sqrt{1 - x^2}$, and so there is no explicit function for y in terms of x. As such we cannot directly compute an expression for the derivative of y with respect to x. However, in a local neighbourhood around a given (x, y) pair, we can describe a single-valued function $y(x)$ and compute its gradient at (x, y) using implicit differentiation as

$$\frac{\mathrm{d}y}{\mathrm{d}x} = -\left(\frac{\partial f}{\partial y}\right)^{-1}\left(\frac{\partial f}{\partial x}\right) \qquad \text{(from Theorem 3.1)} \tag{99}$$

$$= -\left(\frac{1}{2y}\right)(2x) \tag{100}$$

$$= -\frac{x}{y} \tag{101}$$

where in the second line we have substituted the corresponding partial derivatives of $f(x, y) = x^2 + y^2 - 1$. In this trivial example we can compare against splitting the circle into two pieces—a top half and a bottom half—with an

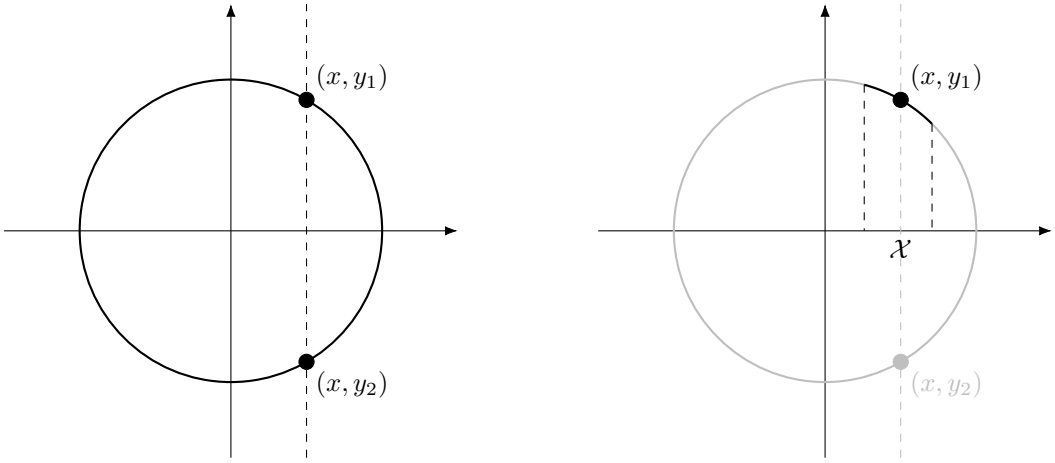


Figure 29: Illustration of Dini's implicit function theorem for a unit circle, $f(x, y) = x^2 + y^2 - 1$. Here at any point x we have two solutions, $y_1(x) = \sqrt{1 - x^2}$ and $y_2(x) = -\sqrt{1 - x^2}$.

explicit expression for each piece, $y = \pm\sqrt{1 - x^2}$, and directly differentiating each as

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}}{\mathrm{d}x} \pm \sqrt{1 - x^2} \tag{102}$$

$$= \frac{\mp 2x}{2\sqrt{1 - x^2}} \tag{103}$$

$$= -\frac{x}{y} \qquad \text{(substituting } y = \pm\sqrt{1 - x^2}\text{)}, \tag{104}$$

i.e., the same result. Note that the function is not differentiable at $x = \pm 1$ (which gives $y = 0$).

3.8 Differentiating Unconstrained Optimisation Problems

Getting back to the question of how to differentiate the solution to an optimisation problem. To warm up we start with the case of unconstrained optimisation. Let $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ be a twice differentiable function and let

$$y(x) \in \operatorname{argmin}_{u \in \mathbb{R}^m} f(x, u) \tag{105}$$

then for non-zero Hessian we have

$$\frac{\mathrm{d}y(x)}{\mathrm{d}x} = -\left(\frac{\partial^2 f}{\partial y^2}\right)^{-1} \frac{\partial^2 f}{\partial x \partial y}. \tag{106}$$

Here $\frac{\partial^2 f}{\partial y^2}$ and $\frac{\partial^2 f}{\partial x \partial y}$ denote matrices of partial second derivatives, i.e., $\left(\frac{\partial^2 f}{\partial y^2}\right)_{ij} = \frac{\partial^2 f}{\partial y_i \partial y_j}$ and $\left(\frac{\partial^2 f}{\partial x \partial y}\right)_{ij} = \frac{\partial^2 f}{\partial y_i \partial x_j}$.

The result is quite easy to prove and follows directly from Dini's implicit function theorem applied to the first-order optimality condition for differentiable unconstrained problems as the following proof shows.[23]

Proof. The first-order optimality condition states that the derivative of f vanishes at (x, y), i.e., $y \in \operatorname{argmin}_u f(x, u)$ implies that $\frac{\partial f(x,y)}{\partial y} = 0$. Now differentiating each side of the optimality condition with respect to x we have

$$\frac{\mathrm{d}}{\mathrm{d}x}\frac{\partial f(x, y)}{\partial y} = \frac{\partial^2 f(x, y)}{\partial x \partial y} + \frac{\partial^2 f(x, y)}{\partial y^2}\frac{\mathrm{d}y}{\mathrm{d}x} \qquad \text{(LHS)}$$

$$\frac{\mathrm{d}}{\mathrm{d}x}0 = 0 \qquad \text{(RHS)}$$

So

$$\frac{\partial^2 f(x, y)}{\partial x \partial y} + \frac{\partial^2 f(x, y)}{\partial y^2}\frac{\mathrm{d}y}{\mathrm{d}x} = 0$$

Rearranging gives the result. □

[23] Here we use $\frac{\partial}{\partial x}$ and $\frac{\partial}{\partial y}$ to denote differentiation with respect to the first and second argument of f, respectively.

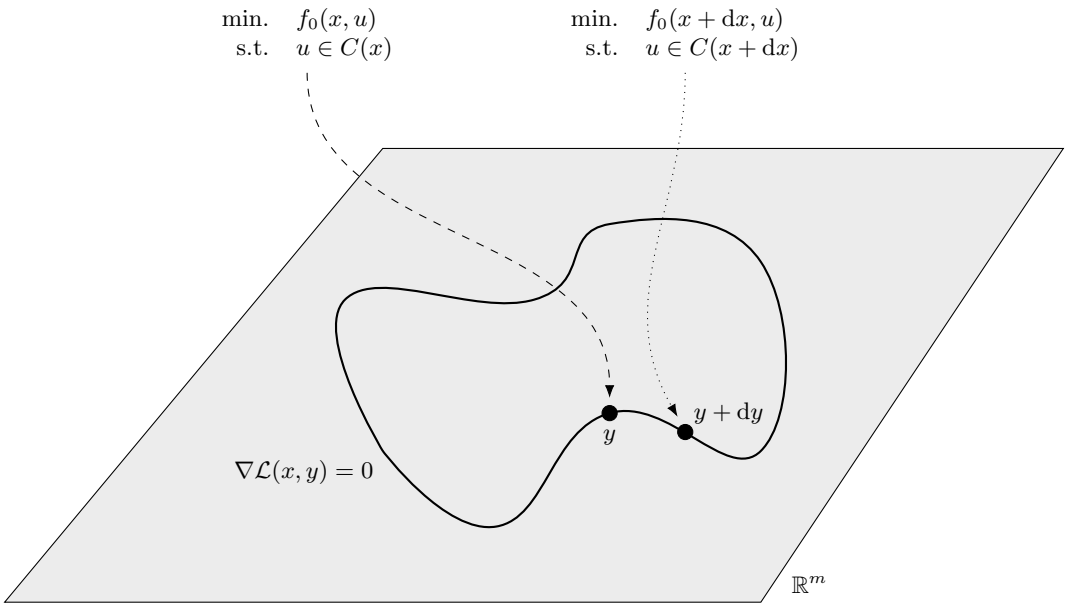


Figure 30: Conceptual view of differentiable optimisation and the optimality condition manifold.

3.9 Differentiating Constrained Optimisation Problems

Before moving on to the case of constrained optimisation problems, let us stop to consider a conceptual view of differentiable optimisation. A cartoon illustration of the high-level idea is shown in Figure 30. Within the output space, $\mathbb{R}^m$ there exists a manifold[24] representing points that satisfy the implicitly defined optimality condition, $\nabla\mathcal{L}(x,y)=0$, for some given x. Solving a parametrized optimisation problem conditioned on x will find a point (perhaps, many) on this manifold. If we change the parameters slightly to, say, $x+\mathrm{d}x$ then the point on the manifold moves a little bit to, say, $y+\mathrm{d}y$. The ratio between the small change in output $\mathrm{d}y$ to the small change in input $\mathrm{d}x$ is the quantity that we are seeking. That is, once on the manifold we can use the implicit function theorem to compute the derivative of y with respect to x.

We now present the result for equality constrained optimisation problems that makes use of the fact that the solution is a stationary point of the Lagrangian associated with the problem.

Theorem 3.2: (Gould et al. [12]). 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

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

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

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

where

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

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

The condition that H is non-singular amounts to the Lagrangian being locally strongly convex in the neighbourhood of (x,y). The condition $\mathbf{rank}\,(A)=q$ is another way of saying that y is a regular point, with regular defined at any feasible point as follows (for problems with both equality and inequality constraints).

[24] This may not be a true manifold in the mathematical sense at non-differentiable points of the solution mapping.

Definition 3.3: (Regular Point [4]). A feasible point u is said to be *regular* if the equality constraint gradients $\frac{\mathrm{d}}{\mathrm{d}u}h_i(x,u)$ and the active inequality constraint gradients $\frac{\mathrm{d}}{\mathrm{d}u}f_i(x,u)$ are linearly independent, or there are no equality constraints and the inequality constraints are all inactive at u.

The proof of the above theorem is fairly straightforward albeit requiring some careful linear algebra.

Proof. Forming the Lagrangian at optimal y for fixed x we have

$$\mathcal{L}(x,y,\nu) = f(x,y) - \sum_{i=1}^{q} \nu_i h_i(x,y).$$

Since $\frac{\partial h(x,y)}{\partial y}$ is full rank we have that y is a regular point. Then there exists a ν such that the Lagrangian is stationary at the point (y,ν). Thus

$$\begin{bmatrix} \frac{\partial \mathcal{L}}{\partial Y}^T \\ \frac{\partial \mathcal{L}}{\partial \nu}^T \end{bmatrix} = \begin{bmatrix} \left(\frac{\partial f(x,y)}{\partial y} - \sum_{i=1}^{q} \nu_i \frac{\partial h_i(x,y)}{\partial y}\right)^T \\ h(x,y) \end{bmatrix} = \mathbf{0}_{m+q}$$

which we can differentiate with respect to x,

$$\frac{\mathrm{d}}{\mathrm{d}x}\begin{bmatrix} \left(\frac{\partial f(x,y)}{\partial y}\right)^T - \sum_{i=1}^{q} \nu_i \left(\frac{\partial h_i(x,y)}{\partial y}\right)^T \\ h(x,y) \end{bmatrix} = \mathbf{0}_{(m+q)\times n}$$

to get, after some re-arranging to put in matrix form,

$$\begin{bmatrix} \frac{\partial^2 f(x,y)}{\partial y^2} - \sum_{i=1}^{q} \nu_i \frac{\partial^2 h_i(x,y)}{\partial y^2} & -\left(\frac{\partial h(x,y)}{\partial y}\right)^T \\ \frac{\partial h(x,y)}{\partial y} & \mathbf{0}_{q\times q} \end{bmatrix} \begin{bmatrix} \frac{\mathrm{d}y(x)}{\mathrm{d}x} \\ \frac{\mathrm{d}\nu(x)}{\mathrm{d}x} \end{bmatrix} = -\begin{bmatrix} \frac{\partial^2 f(x,y)}{\partial x \partial y} - \sum_{i=1}^{q} \nu_i \frac{\partial^2 h_i(x,y)}{\partial x \partial y} \\ \frac{\partial}{\partial x} h(x,y) \end{bmatrix}$$

$$\begin{bmatrix} H & -A^T \\ A & \mathbf{0}_{q\times q} \end{bmatrix} \begin{bmatrix} \frac{\mathrm{d}y(x)}{\mathrm{d}x} \\ \frac{\mathrm{d}\nu(x)}{\mathrm{d}x} \end{bmatrix} = -\begin{bmatrix} B \\ C \end{bmatrix}$$

where in the last line we have substituted quantities A, B, C and H from the theorem statement.[25] We can solve this system of equations directly or solve more efficiently by variable elimination. Multiplying out we have

$$H\frac{\mathrm{d}y(x)}{\mathrm{d}x} - A^T\frac{\mathrm{d}\nu(x)}{\mathrm{d}x} = -B \tag{a}$$

$$A\frac{\mathrm{d}y(x)}{\mathrm{d}x} = -C \tag{b}$$

From (a) we have

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

which substituting into (b) gives,

$$AH^{-1}(A^T\frac{\mathrm{d}\nu(x)}{\mathrm{d}x} - B) = -C$$

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

Then substituting back into (a) we get the result

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

□

The expression in the theorem simplifies if the optimisation problem has specific properties. For example, if the constraints do not depend on the input x then term C disappears. And if the constraints are linear then we can ignore

[25] This should remind you of the KKT system of equations used in Newton's method [6, §10.1].

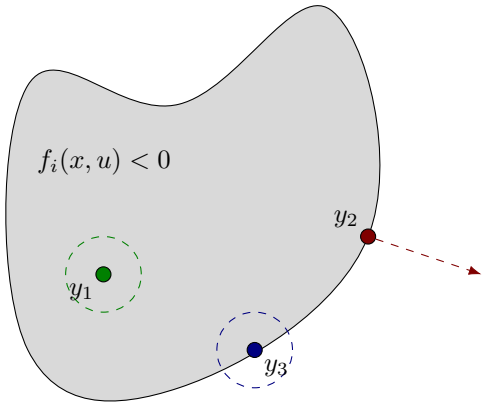


Figure 31: Differentiable optimisation with inequality constraints.

terms involving the Lagrange multipliers ν_i. In the next lecture we will see an example of exploiting the structure of the objective and constraint functions to obtain very efficient backward pass code.

The theorem can also be extended to handle singular H. In particular, as evident when looking through the above proof, a valid gradient can be computed using pseudo-inverses if

$$\begin{bmatrix} B \\ C \end{bmatrix} \in \mathbf{range}\left(\begin{bmatrix} H & -A^T \\ A & 0 \end{bmatrix}\right) \tag{107}$$

and $\mathbf{null}(H) \cap \mathbf{null}(A) = \{0\}$.

We need to be a little more careful in dealing with problems that contain inequality constraints as the problems are non-differentiable since, roughly speaking, constraints change from being active to inactive, or vice versa, when moving from x to $x + \mathrm{d}x$. Technically, this is because the Lagrange multipliers associated with the inequality constraint functions must be non-negative. For Lagrange multipliers which are zero, the gradient becomes one-sided (the variable can increase in value but not decrease).

The situation is illustrated in Figure 31 for a single inequality constraint function f_i. Let λ_i be the Lagrange multiplier associated with f_i. There are three cases that we need to consider:

- If $f_i(x, y) < 0$, then we must have $\lambda_i = 0$. But in this case it is safe to ignore the inequality altogether (for convex optimisation problems);
- If $f_i(x, y) = 0$ and $\lambda_i > 0$, then we can treat the inequality constraint as an equality constraint and apply the result from above;
- If $f_i(x, y) = 0$ and $\lambda_i = 0$, then the problem is not differentiable since moving x in one direction will result in a strictly feasible solution and moving in the other direction will result in infeasibility. In deep learning it is often okay to simply sweep this issue under the rug and ignore the inequality.[26]

An alternative for dealing with inequality constraints is to eliminate them from the problem by using the log-barrier approximation as we discussed in the last lecture. Note, however, that this makes assumptions about how the problem was solved and may cause difficulties for solutions where an inequality is tight, $f_i(x, y) = 0$, thereby making the log-barrier undefined, $\log(-f_i(x, y)) = \log(0)$.

3.10 Automatic Differentiation for Differentiable Optimisation

We have already seen that we can (sometimes) back propagate through the optimisation algorithm used to solve lower-level problems in bi-level optimisation. This can also be used for the backward pass of declarative nodes, and should be the first thing to try, always. At the other extreme we can hand-craft code to compute the gradients using the implicit differentiation results discussed above. This often results in the most efficient implementation but can be laborious and error-prone [13] so should only be done once it has been determined that the declarative node is the bottleneck (or back-propagation through the optimisation algorithm is not possible).

Between these two extremes we can use automatic differentiation to compute the necessary quantities A, B, C and H, needed for $\frac{\mathrm{d}y}{\mathrm{d}x}$ given software implementations of the objective function f and constraint functions h. These quantities are, after all, just first and second partial derivatives. Another option is to derive and implement the optimality conditions in software, and then use automatic differentiation on that code. We won't explore these two options further in these lectures. The interested reader is referred to Blondel et al. [5].

[26] A similar situation occurs when computing the gradient of the popular ReLU function when its argument is zero.

3.11 Vector-Jacobian Product

An important consideration when implementing declarative nodes is the order in which we evaluate terms in the expressions for the derivatives. This can have an enormous impact on the efficiency of back-propagation during the deep learning backward pass. We will see later that problem specific simplifications, and calculation re-use, can also lead to significant computational savings. To illustrate this, consider back-propagating through the generic unconstrained optimisation case with output variable $y \in \mathbb{R}^m$ and input variable $x \in \mathbb{R}^n$. The backward pass computes

$$\frac{\mathrm{d}L}{\mathrm{d}x} = \frac{\mathrm{d}L}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}x} = \underbrace{\left(v^T\right)}_{\mathbb{R}^{1\times m}}\underbrace{\left(-H^{-1}B\right)}_{\mathbb{R}^{m\times n}} \tag{108}$$

where $v^T = \frac{\mathrm{d}L}{\mathrm{d}y}$ is the incoming gradient of the loss with respect to the output.

Let us assume that H^{-1} is already factored (taking $O(m^3)$ if unstructured or less if structured). There are two ways we can evaluate Equation 108. First, we can evaluate from right-to-left, i.e., $-v^T\left(H^{-1}B\right)$. Here we pay $O(m^2n+mn)$ operations. Second, we can evaluate from left-to-right, i.e., $\left(-v^TH^{-1}\right)B$. This ordering only costs $O(m^2+mn)$, i.e., approximately n times less work, and so is preferred.

3.12 Summary and Open Questions

In this lecture we showed how (continuous) optimisation problems can be embedded *inside* deep learning models resulting in a bi-level optimisation problem for training the model. Back-propagation can be achieved by either unrolling the optimisation algorithm used to solve the problem in the forward pass or via implicit differentiation of the problem's optimality conditions. The former is easy to implement using automatic differentiation but memory intensive since intermediate calculations need to be stored in the forward pass as well as allocation for buffers to store gradients for the chain rule in the backward pass.[27]

The latter, implicit differentiation, requires that we have strong convexity in the neighbourhood of the solution (for invertibility of H). It is also more effort to implement but on the other hand does not need to know how the problem was solved (in the forward pass), nor store any intermediate calculations, so has the advantage that specialised solvers can be used. In particular, non-differentiable steps may be used when solving the problem without affecting our ability to compute gradients in the backward pass. The biggest drawback is perhaps the need to compute H^{-1}, which may be costly.[28] We will see in the next lecture that for many problems H has structure that we can exploit for more efficient numerical calculations.

The area of differentiable optimisation, especially as it applies to deep learning (and meta-learning—see, e.g., [18, 20]), is very active and there are many open questions:

- Are declarative nodes slower? Since we need to solve an optimisation problem for each instance in the forward pass (and at test time), it may not even be desirable to include optimisation problems within a deep learning model in the first place independent of how quickly we can perform the backward pass or how much time we're willing to devote to training.
- Do declarative nodes give theoretical guarantees? A large literature on optimisation exists that may allow us to provide some guarantees on the output of declarative nodes. However, since these nodes are still wrapped inside a larger generic deep learning model such guarantees might be lost end-to-end.
- As already mentioned the need to invert H is perhaps the biggest drawback for computing exact gradients. Can H^{-1} be approximated to reduce computational cost in the case where structure cannot be exploited? One promising line of research uses a Neumann series to efficiently approximate v^TH^{-1} and show good results for large-scale hyperparameter search [22].
- How best to handle non-smooth or discrete optimization problems? Even linear programs have zero gradient almost everywhere so tricks have to be applied in order to use them within end-to-end learning frameworks. Some recent work on generalized Clarke gradients, perturb/regularised optimizers and linear relaxations to discrete problems is showing promise here. See, for example, Berthet et al. [3] and Vlastelica et al. [36].
- What happens when the best-response function is a set-valued function, i.e., when problems have multiple solutions? Even convex problems can have a (dense) set of solutions resulting in H being singular. Here Toso et al. [34] suggest a trust region method for stability and guaranteeing non-singularity of H. For non-convex problems finding different solutions during different iterations may confuse the learning algorithm. Gould et al.

[27] Those familiar with deep learning literature will notice the similarity here with recurrent neural networks (e.g., RNNs and LSTMs) where gradient calculations on the unrolled network is often called "back-propagation through time."

[28] In code we should never explicitly invert H. Always use a solver (i.e., factor then solve) instead, which gives better numerical precision.

[12] has a toy example. In the bi-level optimisation literature, researchers talk about optimistic (weak) and pessimistic (strong) formulations [20] where the decision made by the follower is a best case or worst case for the leader, respectively, although these too may be set-valued.

- What if the forward pass solution is suboptimal? Or more importantly, can we save compute by only solving a problem approximately and still get a descent direction for learning? What happens at test time?
- Can problems become infeasible during learning? For example, if the feasible set is parametrized, it may become empty at some point during training. Is there a way to guard against this, say, through different parametrizations?

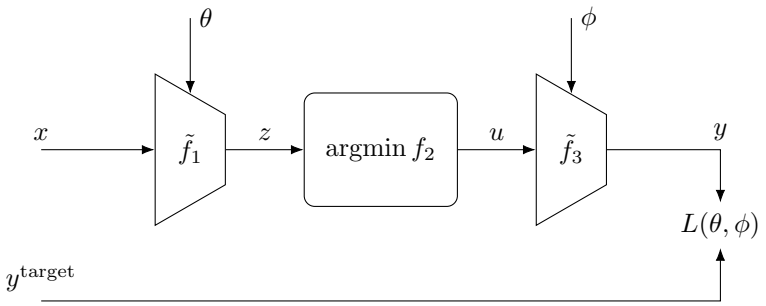


Figure 32: A typical architecture for training a model containing differentiable optimisation.

4 Examples and Applications

This lecture is based on material from the deep declarative networks GitHub repository, available at `http://deepdeclarativenetworks.com`. Further online resources are provided at the end of the lecture.

Prior to knowing how to differentiate through optimisation problems, many researchers used optimisation as a post-processing step on pre-trained networks, or as a pre-processing step on raw data before training. Both of these are still common practice. However, with the ability to propagate gradients through the solution to an optimisation problem we can explore many more exciting directions. A typical application may look something like that shown in Figure 32. Here raw data x is pre-processed by a standard imperative deep learning network to produce input z for a declarative node. The declarative node solves some optimisation problem producing a solution u, which is then optionally post-processed (by another standard imperative deep learning network) to give the final model output y. The pre-processing and post-processing networks may contain learnable parameters, θ and ϕ, respectively. Importantly, these parameters can all be learned end-to-end given a loss applied to the output y and ground-truth target y^{target} (and any other losses applied to intermediate outputs along the way). Of course, multiple declarative nodes and arbitrary network configurations are also possible.

We now present example declarative nodes, derive their gradients, and profile their backward pass implementations.

4.1 Least Squares

Let us start with our old friend, the least-squares problem,

$$\text{minimize} \quad \|Ax - b\|_2^2 \tag{109}$$

with closed-form solution $x^\star = \left(A^TA\right)^{-1}A^Tb$ where $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, and A is full rank. We are interested in computing the gradient of the solution with respect to the elements of A.[29] Differentiating $x^\star$ with respect to single element A_{ij}, we have

$$\frac{\mathrm{d}}{\mathrm{d}A_{ij}}x^\star = \frac{\mathrm{d}}{\mathrm{d}A_{ij}}\left(A^TA\right)^{-1}A^Tb \tag{110}$$

$$= \left(\frac{\mathrm{d}}{\mathrm{d}A_{ij}}\left(A^TA\right)^{-1}\right)A^Tb + \left(A^TA\right)^{-1}\left(\frac{\mathrm{d}}{\mathrm{d}A_{ij}}A^Tb\right). \tag{111}$$

Now using the identity $\frac{\mathrm{d}}{\mathrm{d}z}Z^{-1} = -Z^{-1}\left(\frac{\mathrm{d}}{\mathrm{d}z}Z\right)Z^{-1}$ we get, for the first term,

$$\frac{\mathrm{d}}{\mathrm{d}A_{ij}}\left(A^TA\right)^{-1} = -\left(A^TA\right)^{-1}\left(\frac{\mathrm{d}}{\mathrm{d}A_{ij}}\left(A^TA\right)\right)\left(A^TA\right)^{-1} \tag{112}$$

$$= -\left(A^TA\right)^{-1}\left(E_{ij}^TA + A^TE_{ij}\right)\left(A^TA\right)^{-1} \tag{113}$$

where E_{ij} is a matrix with one in the (i,j)-th element and zeros elsewhere. Furthermore, for the second term,

$$\frac{\mathrm{d}}{\mathrm{d}A_{ij}}A^Tb = E_{ij}^Tb \tag{114}$$

[29] We could also consider differentiating $x^\star$ with respect to the elements of b, which is trivially $\frac{\mathrm{d}}{\mathrm{d}b}x^\star = \left(A^TA\right)^{-1}A^T$.

Plugging these back into Equation 111 we have

$$\frac{\mathrm{d}}{\mathrm{d}A_{ij}}x^\star = -\left(A^TA\right)^{-1}\left(E_{ij}^TA + A^TE_{ij}\right)\left(A^TA\right)^{-1}A^Tb + \left(A^TA\right)^{-1}E_{ij}^Tb \tag{115}$$

$$= -\left(A^TA\right)^{-1}\left(E_{ij}^TA + A^TE_{ij}\right)x^\star + \left(A^TA\right)^{-1}E_{ij}^Tb \tag{116}$$

$$= -\left(A^TA\right)^{-1}\left(E_{ij}^T(Ax^\star - b) + A^TE_{ij}x^\star\right) \tag{117}$$

$$= -\left(A^TA\right)^{-1}\left((a_i^Tx^\star - b_i)e_j + x_j^\star a_i\right) \tag{118}$$

where $e_j = (0, 0, \ldots, 1, 0, \ldots) \in \mathbb{R}^n$ is the j-th canonical vector, i.e., vector with a one in the j-th component and zeros everywhere else, and $a_i^T \in \mathbb{R}^{1\times n}$ is the i-th row of matrix A.

Observe that the term $(A^TA)^{-1}$ appears both in the solution for x and the derivatives with respect to each A_{ij}. Thus, it only needs to be computed (or more precisely factored into, say, Q and R) during the forward pass and stored for use in the backward pass. This saves significant compute. Moreover, we can reuse terms for different $\frac{\mathrm{d}}{\mathrm{d}A_{ij}}$ and efficiently combine with the incoming gradient of the loss with respect to $x^\star$ as we now show.

Let $r = b - Ax^\star$ and let v^T denote the backward coming gradient $\frac{\mathrm{d}L}{\mathrm{d}x^\star}$. Then

$$\frac{\mathrm{d}L}{\mathrm{d}A_{ij}} = v^T\frac{\mathrm{d}x^\star}{\mathrm{d}A_{ij}} \tag{119}$$

$$= v^T\left(A^TA\right)^{-1}\left(r_ie_j - x_j^\star a_i\right) \tag{120}$$

$$= w^T\left(r_ie_j - x_j^\star a_i\right) \tag{121}$$

$$= r_iw_j - w^Ta_ix_j^\star \tag{122}$$

where $w = \left(A^TA\right)^{-1}v$ does not depend on which A_{ij} we are differentiating with respect to. We can therefore compute the entire matrix of $m \times n$ derivatives efficiently as the sum of two outer products

$$\left(\frac{\mathrm{d}L}{\mathrm{d}A}\right)^T = \left[\frac{\mathrm{d}L}{\mathrm{d}A_{ij}}\right]_{\substack{i=1,\ldots,m\\ j=1,\ldots,n}} = rw^T - (Aw)(x^\star)^T \tag{123}$$

PyTorch code for the forward and backward pass operating on batched data is shown below.[30]

```
class LeastSquaresFcn(torch.autograd.Function):
    """PyTorch autograd function for least squares, minimize_{x} \|Ax - b\|."""

    @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

    @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,
```

[30] Deep learning frameworks process data in batches passed as tensors. In PyTorch, the first dimension of the tensor is the batch dimension, and many built-in methods are batch-aware.

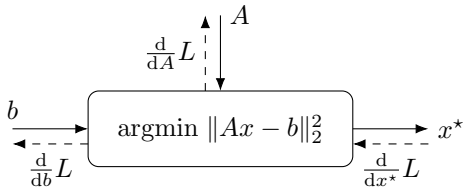


Figure 33: Least-squares declarative node with input b and parameters A.

```
32              torch.linalg.solve_triangular(torch.transpose(R, 2, 1), dx, upper=False), upper=True)
33          Aw = torch.bmm(A, w)
34
35          if ctx.needs_input_grad[0]:
36              r = b - torch.bmm(A, x)
37              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))
38          if ctx.needs_input_grad[1]:
39              db = Aw
40
41          # return gradients
42          return dA, db
```

Here we use QR factorisation to solve for $x^\star$ and efficiently compute the gradient in the backward pass. Let $A = QR$. Then, from Equation 19, we have

$$x^\star = R^{-1}Q^T b \tag{124}$$

and, similarly,

$$w = (A^T A)^{-1} v = R^{-1}R^{-T}v. \tag{125}$$

The forward pass implementation starts on Line 5. Since we are overriding the backward pass function (i.e., not using automatic differentiation), we tell PyTorch that it does not need to track calculations in the forward pass (Line 9). Matrix A is factored into Q and R on Line 10, which are then used to solve for $x^\star$ on Line 11. This essentially completes the forward pass, with the input and intermediate calculations cached (Line 14) for use in the backward pass. The backward pass, starting on Line 20, first retrieves the intermediate calculations from the forward pass (Line 26) and then performs some housekeeping. On Line 31 we compute $w = R^{-1}R^{-T}v$ by solving two successive systems of linear equations, both of which are triangular. We then compute Aw on Line 33, taking into account that the data is provided in batches. Lines 36 and 37 complete the calculation for $\frac{\mathrm{d}}{\mathrm{d}A}L$ making use of the `bmm` method to efficiently compute outer products wr^T and $x^\star(Aw)^T$ on batch data.

We conduct an experiment to evaluate the running time and memory use for three different implementations of differentiable least-squares: (1) using PyTorch's built-in `lstsq` method, (2) automatic differentiation on the solution via QR factorisation, and (3) our custom backward pass from above. To profile the code we set up a bi-level optimisation problem with least-squares as the lower-level problem,

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\|x^\star - x^{\text{target}}\|_2^2 \\ \text{subject to} & x^\star = \operatorname{argmin}_x \|Ax - b\|_2^2 \end{array} \tag{126}$$

A schematic of least-squares as a declarative node is shown in Figure 33, and is essentially the entire network for this toy example. The upper-level problem attempts to find a matrix A that produces the given solution x^{target} from the lower-level problem. We run on randomly initialised matrices $A \in \mathbb{R}^{m \times n}$ and target vectors $x^{\text{target}} \in \mathbb{R}^n$ for 1000 iterations of gradient descent for various problem sizes from $n = 10$ to $n = 100$ (and $m = 2n$). Results are shown in Figure 34. Our custom implementation is clearly faster and more memory efficient than the automatic differentiation approaches.

4.2 Optimal Transport

Optimal transport is a very popular technique in machine learning with applications from measuring distances between probability distributions to loss functions for training GANs. One way to think of optimal transport is as performing a mapping from an m-by-n dimensional real cost matrix, denoted M, to an m-by-n dimensional probability matrix, denoted P, whose entries represents the probability of matching between two sets of items, i.e., the item in the i-th row to the item in the j-th column.

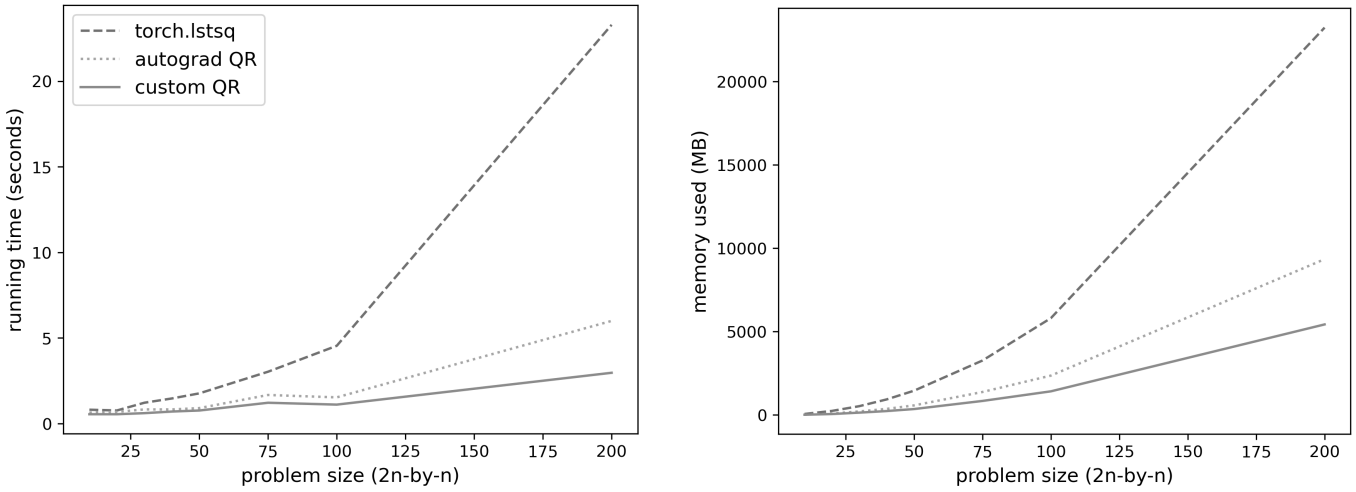


Figure 34: Running time and memory usage for differentiable least squares versus problem size.

$$\begin{array}{ll} \text{minimize} & \langle M, P\rangle + \frac{1}{\gamma}\langle P, \log P\rangle \\ \text{subject to} & P\mathbf{1} = r \\ & P^T\mathbf{1} = c \end{array} \tag{127}$$

usually with $\mathbf{1}^T r = \mathbf{1}^T c = 1$, and hence $\mathbf{1}^T P \mathbf{1} = 1$.

It can be shown [8] that the solution to the entropy regularized optimal transport problem takes the following form,

$$P_{ij} = \alpha_i \beta_j e^{-\gamma M_{ij}} \tag{128}$$

Moreover, the optimal transport problem is easily solved using the Sinkhorn algorithm, which repeatedly performs row and column normalisations converging to a matrix in the feasible set. Specifically, initialising $K_{ij} = e^{-\gamma M_{ij}}$ and $\alpha, \beta \in \mathbb{R}^n_{++}$ the Sinkhorn algorithm iterates over the following two steps

$$\alpha \leftarrow r \oslash K\beta \tag{129}$$

$$\beta \leftarrow c \oslash K^T\alpha, \tag{130}$$

where $\oslash$ denotes componentwise division, until convergence.[31] Then the solution to the optimal transport problem is $P = \mathbf{diag}(\alpha) K \mathbf{diag}(\beta)$.

Because this algorithm involves only differentiable steps it can be unrolled, and automatic differentiation applied. Alternatively, since the algorithm solves an optimisation problem, we can apply the deep declarative network result, using implicit differentiation to calculate gradients in the backward pass.

Our derivation of gradients in the backward pass follows Gould et al. [13]. Let us start by writing down the objective function $f : \mathbb{R}^{m\times n} \times \mathbb{R}^{m\times n} \to \mathbb{R}$ and (vector-valued) constraint function $h : \mathbb{R}^{m\times n} \times \mathbb{R}^{m\times n} \to \mathbb{R}^{m+n}$,

$$f(M, P) = \sum_{i=1}^{m}\sum_{j=1}^{n} M_{ij}P_{ij} + \frac{1}{\gamma}\sum_{i=1}^{m}\sum_{j=1}^{n} P_{ij}\log P_{ij} \tag{131}$$

$$h(M, P) = \begin{bmatrix} \mathbf{1}_n^T & \mathbf{0}_n^T & \dots & \mathbf{0}_n^T \\ \mathbf{0}_n^T & \mathbf{1}_n^T & \dots & \mathbf{0}_n^T \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0}_n^T & \mathbf{0}_n^T & \dots & \mathbf{1}_n^T \\ I_{n\times n} & I_{n\times n} & \dots & I_{n\times n} \end{bmatrix} \begin{bmatrix} P_{11} \\ P_{12} \\ \vdots \\ P_{1n} \\ P_{21} \\ \vdots \\ P_{mn} \end{bmatrix} - \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_m \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \mathbf{0}_{m+n}. \tag{132}$$

Note that the set of constraint functions is redundant: if any subset of $m + n - 1$ constraints are satisfied then the remaining constraint will also be satisfied. This can be seen by observing that any row within the matrix of coefficients

[31] In matrix notation we can write $a \oslash b$ as $\mathbf{diag}(b)^{-1}\, a$.

can be written as a linear combination of the other rows. For example, the first row is the sum of the last n rows minus the sum of rows 2 to m. While this is not a problem for the forward pass optimisation it will cause issues when applying Theorem 3.2, which requires $\frac{\mathrm{d}h}{\mathrm{d}P}$ to be full rank, during the backward pass. The problem is easily rectified by removing the first row,

$$h(M,P) = \begin{bmatrix} \mathbf{0}_n^T & \mathbf{1}_n^T & \dots & \mathbf{0}_n^T \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0}_n^T & \mathbf{0}_n^T & \dots & \mathbf{1}_n^T \\ I_{n\times n} & I_{n\times n} & \dots & I_{n\times n} \end{bmatrix} \begin{bmatrix} P_{11} \\ P_{12} \\ \vdots \\ P_{1n} \\ P_{21} \\ \vdots \\ P_{mn} \end{bmatrix} - \begin{bmatrix} r_2 \\ \vdots \\ r_m \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \mathbf{0}_{m+n-1}. \tag{133}$$

To simplify notation we will think of matrices M and P being represented in vectorised form so that objective and constraint functions become $f : \mathbb{R}^{mn} \times \mathbb{R}^{mn} \to \mathbb{R}$ and $h : \mathbb{R}^{mn} \times \mathbb{R}^{mn} \to \mathbb{R}^{m+n-1}$, respectively, i.e., as vectors of length mn rather than matrices of size $m \times n$. This can already be seen in how we wrote the equations above. As a reminder we need to compute,

$$\frac{\mathrm{d}P}{\mathrm{d}M} = \left(H^{-1}A^T\left(AH^{-1}A^T\right)^{-1}AH^{-1} - H^{-1}\right)B \tag{134}$$

We have, by direct examination of the constraint function h,

$$A = \frac{\mathrm{d}h}{\mathrm{d}P} \in \mathbb{R}^{(m+n-1)\times mn} \tag{135}$$

$$= \begin{bmatrix} \mathbf{0}_n^T & \mathbf{1}_n^T & \dots & \mathbf{0}_n^T \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0}_n^T & \mathbf{0}_n^T & \dots & \mathbf{1}_n^T \\ I_{n\times n} & I_{n\times n} & \dots & I_{n\times n} \end{bmatrix} \tag{136}$$

For terms H and B we have,

$$\frac{\partial f}{\partial P_{ij}} = M_{ij} + \frac{1}{\gamma}\log P_{ij} + \frac{1}{\gamma} \tag{137}$$

and therefore

$$H_{ij,kl} = \frac{\partial^2 f}{\partial P_{ij}\partial P_{kl}} = \begin{cases} \frac{1}{\gamma P_{ij}} & \text{if } ij = kl \\ 0 & \text{otherwise} \end{cases} \tag{138}$$

and

$$B_{ij,kl} = \frac{\partial^2 f}{\partial P_{ij}\partial M_{kl}} = \begin{cases} 1 & \text{if } ij = kl \\ 0 & \text{otherwise.} \end{cases} \tag{139}$$

Observe that these are diagonal matrices, which makes sense since the objective does not couple terms. Moreover, matrix B is an mn-times-mn identity matrix, so can be ignored for the purposes of evaluating matrix expressions. The matrix H being diagonal is trivially invertible, namely,

$$H^{-1} = \gamma\,\mathbf{diag}(P_{ij} \mid i = 1,\dots,m; j = 1,\dots,n)\,. \tag{140}$$

The only term left to figure out is $(AH^{-1}A^T)^{-1}$. We can directly write out the (k,l)-th entry of $AH^{-1}A^T$ for $k,l \in 1,\dots,m+n-1$ as

$$\begin{aligned}(AH^{-1}A^T)_{kl} &= \sum_{i=1}^{m}\sum_{j=1}^{n}\frac{A_{k,ij}A_{l,ij}}{H_{ij,ij}} \\ &= \gamma\sum_{i=1}^{m}\sum_{j=1}^{n}A_{k,ij}A_{l,ij}P_{ij}\end{aligned}$$

Here we can imagine taking the elementwise product between the k-th and l-th rows of A, and vectorised P, and then summing the results as illustrated below,

$$\begin{matrix} k \\ \\ \\ l \\ \\ \\ \end{matrix} \begin{bmatrix} \mathbf{0}_n^T & \mathbf{1}_n^T & \dots & \mathbf{0}_n^T \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0}_n^T & \mathbf{0}_n^T & \dots & \mathbf{1}_n^T \\ \\ I_{n\times n} & I_{n\times n} & \dots & I_{n\times n} \\ \\ \end{bmatrix} \left.\begin{matrix} \\ \\ \\ \\ \\ \\ \end{matrix}\right\} \begin{matrix} \uparrow \\ m-1 \\ \downarrow \\ \\ \uparrow \\ n \\ \downarrow \end{matrix} \tag{141}$$

$$\underbrace{\begin{bmatrix} P_{1,1:n} & P_{2,1:n} & \dots & P_{m,1:n} \end{bmatrix}}_{\leftarrow \quad mn \quad \rightarrow}$$

Depending on whether k and l both fall in the top block, bottom block or different blocks of A will have different results on the elementwise product and summation. For example, the elementwise product between rows $k \neq l$ both in the top block is always zero. Analysing all four cases, we have for $(AH^{-1}A^T)_{kl} = \gamma \sum_{i=1}^m \sum_{j=1}^n A_{k,ij} A_{l,ij} P_{ij}$,

	$1 \leq l \leq m-1$	$m \leq l \leq m+n-1$
$1 \leq k \leq m-1$	$\begin{cases} \gamma \sum_{j=1}^n P_{k+1,j} & \text{if } k = l \\ 0 & \text{otherwise} \end{cases}$	$\gamma P_{k+1,l-m+1}$
$m \leq k \leq m+n-1$	$\gamma P_{l+1,k-m+1}$	$\begin{cases} \gamma \sum_{i=1}^m P_{i,k-m+1} & \text{if } k = l \\ 0 & \text{otherwise} \end{cases}$

and noting that $\sum_{j=1}^n P_{kj} = r_k$ and $\sum_{i=1}^m P_{ik} = c_k$ we arrive at

$$AH^{-1}A^T = \gamma \begin{bmatrix} \mathbf{diag}(r_{2:m}) & P_{2:m,1:n} \\ P_{2:m,1:n}^T & \mathbf{diag}(c) \end{bmatrix} \tag{142}$$

Now we can directly compute $(AH^{-1}A^T)^{-1}$ in $O((m+n-1)^3)$ time using Cholesky factorization or we can squeeze even more efficiency by making use of block matrix inversion [16], which results in compute of $O((m-1)^3)$ time,[32]

$$\begin{bmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{12}^T & \Lambda_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{diag}(r_{2:m}) & P_{2:m,1:n} \\ P_{2:m,1:n}^T & \mathbf{diag}(c) \end{bmatrix}^{-1}, \tag{143}$$

where each block is calculated as

$$\Lambda_{11} = \left(\mathbf{diag}(r_{2:m}) - P_{2:m,1:n}\mathbf{diag}(c)^{-1}P_{2:m,1:n}^T\right)^{-1} \tag{144}$$

$$\Lambda_{12} = -\Lambda_{11} P_{2:m,1:n}\mathbf{diag}(c)^{-1} \tag{145}$$

$$\Lambda_{22} = \mathbf{diag}(c)^{-1}\left(I - P_{2:m,1:n}^T \Lambda_{12}\right) \tag{146}$$

and we use Cholesky factorization to multiply by positive definite sub-matrix Λ_{11} rather than inverting explicitly.

Example PyTorch source code for efficiently computing the entire backwards gradient

$$\frac{\mathrm{d}L}{\mathrm{d}M} = \frac{\mathrm{d}L}{\mathrm{d}P}\frac{\mathrm{d}P}{\mathrm{d}M} = \gamma v^T \mathbf{diag}(P)\, A^T \begin{bmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{12}^T & \Lambda_{22} \end{bmatrix} A\, \mathbf{diag}(P) - \gamma v^T \mathbf{diag}(P) \tag{147}$$

is shown below (full source code is available at `http://deepdeclarativenetworks.com`). Here, we use the various terms derived above and instead of explicitly multiplying by A, the code makes use of the observation that multiplying the rows of A results in a summation over corresponding elements of the multiplicand as seen in Line 11. Other implementation efficiencies are achieved by keeping gradients in matrix form rather than vectorizing, and observing, for example, that right multiplication by a diagonal matrix scales rows. As such $v^T\mathbf{diag}(P)$ can be implemented as the elementwise product between $\frac{\mathrm{d}L}{\mathrm{d}P}$ and P as is done on Line 8.

[32] Or in $O(n^3)$ time if $n < m$ using an alternative formula for the block inverse.

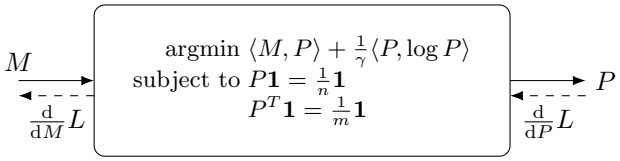


Figure 35: Optimal transport declarative node with input M and output P.

```
@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]) - bmm(P[:,1:m,0:n], P_over_c.transpose(1, 2)))
    lmd_12 = cholesky_solve(P_over_c, lmd_11)
    lmd_22 = diag_embed(1.0 / c) + bmm(lmd_12.transpose(1, 2), P_over_c)

    v1 = torch.cholesky_solve(vHAt1, lmd_11) - torch.bmm(lmd_12, vHAt2)
    v2 = torch.bmm(lmd_22, vHAt2) - torch.bmm(lmd_12.transpose(1, 2), vHAt1)

    # 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
```

As with the least-squares example, we conduct an experiment to evaluate the running time and memory use for different implementations of differentiable optimal transport. We set up a bi-level optimisation problem is a similar fashion, now with optimal transport as the lower-level problem,

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\|P - P^{\text{target}}\|_F^2 \\ \text{subject to} & \text{minimize } \langle M, P\rangle + \frac{1}{\gamma}\langle P, \log P\rangle \\ & \text{subject to} \quad P\mathbf{1} = \frac{1}{n}\mathbf{1} \\ & \qquad\qquad P^T\mathbf{1} = \frac{1}{m}\mathbf{1} \end{array} \tag{148}$$

and upper-level variable $M \in \mathbb{R}^{m\times n}$. A schematic of the optimal transport declarative node is shown in Figure 35.

While the derivation seemed to be a lot of work, the effort was worth it. As can be seen in Figure 36, our implementation (i.e., using implicit differentiation on the optimality conditions) to compute the gradient in the backward pass is much faster than automatically differentiating the Sinkhorn algorithm. However, we need to be smart about constructing and inverting $AH^{-1}A^T$ by using either Cholesky factorization or block Cholesky inversion, since inverting the matrix naively results in slower performance. The implicit differentiation approach is also much more memory efficient than automatic differentiation as can be seen in Figure 37.

4.3 Eigen Decomposition (aka Spectral Decomposition)

Given a real symmetric matrix $X = X^\mathsf{T} \in \mathbb{R}^{m\times m}$, it is well-known that the (unit) eigenvector associated with the largest eigenvalue of X can be found by solving the following equality constrained optimization problem,

$$\begin{array}{ll} \text{maximize (over } u \in \mathbb{R}^m) & u^\mathsf{T} X u \\ \text{subject to} & u^\mathsf{T} u = 1. \end{array} \tag{149}$$

Here we assume that the largest eigenvalue is simple otherwise a well-defined derivative does not exist. The optimality conditions for solution $y \in \mathbb{R}^m$ are thus,

$$Xy - \lambda_{\max} y = 0_m \tag{150}$$

$$y^\mathsf{T} y = 1. \tag{151}$$

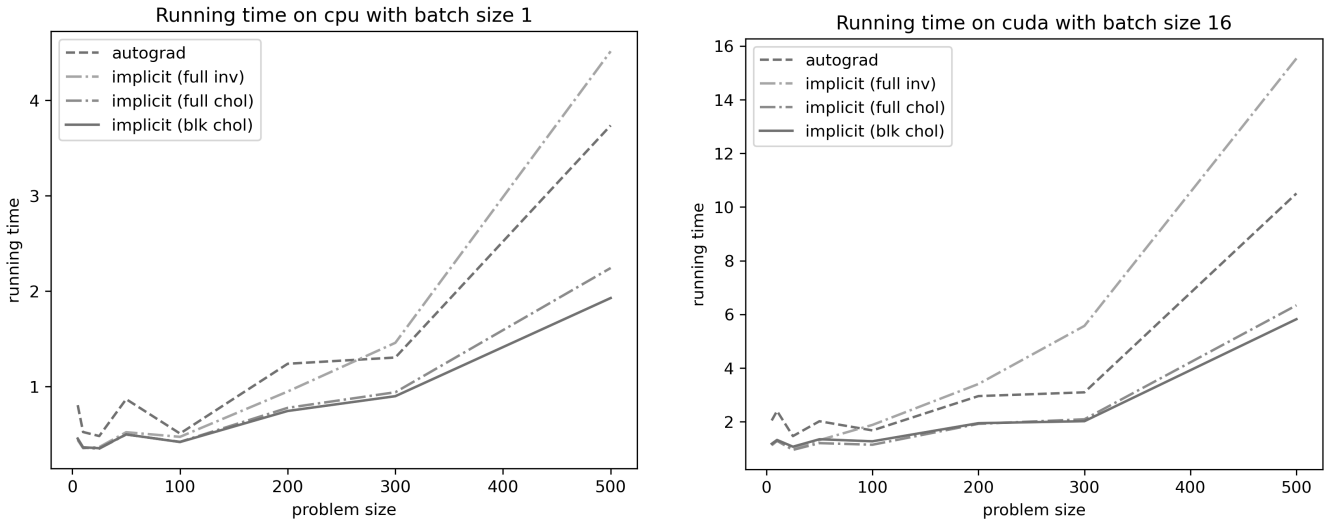
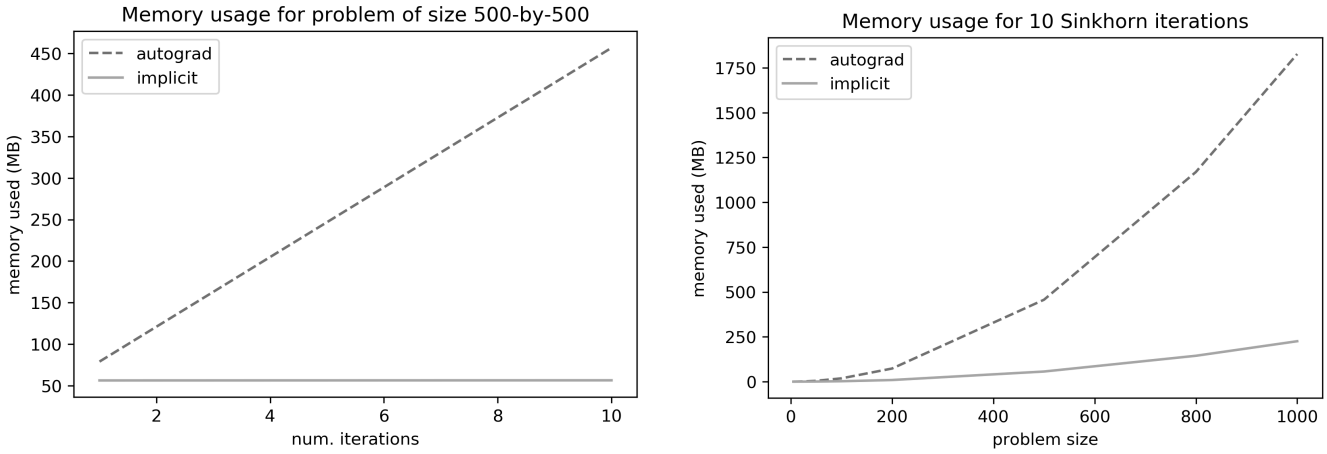


Figure 36: Running time of differentiable optimal transport on CPU (left) and GPU (right) for different problem sizes.



(a) fixed problem size

(b) fixed Sinkhorn iterations

Figure 37: Memory usage of differentiable optimal transport.

Indeed, any eigenvalue-eigenvector pair will also satisfy these conditions (replacing $\lambda_{\max}$ with the appropriate eigenvalue, of course). We can easily extend the above optimization problem to find all eigenvectors (and eigenvalues) of symmetric input matrix X as,

$$\begin{array}{ll} \text{maximize (over } U \in \mathbb{R}^{m\times m}) & \mathbf{tr}\left(U^\mathsf{T} X U\right) \\ \text{subject to} & U^\mathsf{T} U = I_{m\times m}. \end{array} \tag{152}$$

Note also that the solution is not unique, even if all eigenvalues are simple (i.e., even if X has m distinct eigenvalues). Given a solution Y to Problem 152, negating and permuting columns is also a solution. We typically rely on the solver to sort eigenvectors in ascending or descending order corresponding to their eigenvalues. However, the sign ambiguity for each eigenvector is unavoidable.

The forward pass implementation for a differentiable PyTorch function to perform eigen decomposition is shown below. The code uses PyTorch's `eigh` function for computing all eigenvalues and eigenvectors of a symmetric matrix (Line 14), which is equivalent to solving the optimisation problem above.[33]

```
1  class EigenDecompositionFcn(torch.autograd.Function):
2      """PyTorch autograd function for eigen decomposition."""
3
4      # tolerance to consider two eigenvalues equal
5      eps = 1.0e-9
6
7      @staticmethod
8      def forward(ctx, X):
9          B, M, N = X.shape
10         assert N == M
11
12         # use torch's eigh function to find the eigenvalues and eigenvectors of a symmetric matrix
13         with torch.no_grad():
14             lmd, Y = torch.linalg.eigh(0.5 * (X + X.transpose(1, 2)))
15
16         ctx.save_for_backward(X, lmd, Y)
17         return Y
```

Magnus [24] gives the differentials as

$$\mathrm{d}\lambda_k = y_k^\mathsf{T}(\mathrm{d}X)y_k \tag{153}$$

$$\mathrm{d}y_k = (\lambda_k I - X)^\dagger(\mathrm{d}X)y_k \tag{154}$$

for any simple eigenvalue λ_k and it's corresponding eigenvector y_k. Here $(\lambda_k I - X)^\dagger$ is the pseudo-inverse of $(\lambda_k I - X)$, which will be singular since we are zeroing out one of the eigenvalues.[34] So with respect to the (i,j)-th component of X we have

$$\frac{\mathrm{d}y_k}{\mathrm{d}X_{ij}} = -\frac{1}{2}(X - \lambda_k I)^\dagger(E_{ij} + E_{ji})y_k \tag{155}$$

where we have used the fact that X is symmetrical. The same expression can be derived from the deep declarative network results [12].

To compute the gradient of the loss with respect to the (i,j)-th component of X we need to sum over the contributions from all eigenvectors. Let $v_k^\mathsf{T} = \frac{\mathrm{d}}{\mathrm{d}y_k}L \in \mathbb{R}^{1\times m}$. Then,

$$\frac{\mathrm{d}L}{\mathrm{d}X_{ij}} = \sum_{k=1}^{m} v_k^\mathsf{T} \frac{\mathrm{d}y_k}{\mathrm{d}X_{ij}} \tag{156}$$

$$= -\frac{1}{2}\sum_{k=1}^{m} v_k^\mathsf{T}(X - \lambda_k I)^\dagger(E_{ij} + E_{ji})y_k \tag{157}$$

Naively implementing this expression by looping over the contribution from each eigenvector is painfully slow. We show an example of such code below.

[33] You can actually directly back-propagate through the PyTorch `eigh` function being careful to only provide upper- or lower-triangular representations for X so constructing `EigenDecompositionFcn` is not strictly necessary if you want to use eigen decomposition in a deep learning model.

[34] The pseudo-inverse of a symmetric matrix $X = Q\Lambda Q^\mathsf{T}$ can be easily found as $X^\dagger = Q\Lambda^\dagger Q^\mathsf{T}$, where $\Lambda^\dagger$ is constructed by inverting all non-zero diagonal entries of Λ and keeping the zero entries. The derivation of the differentials is mostly straightforward. The one trick is recognising that since $(\lambda_k I - X)y_k = 0$ we also have $(\lambda_k I - X)^\dagger y_k = 0$.

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

    dJdX = torch.zeros_like(X)

    # loop over eigenvalues (version 1)
    for k in range(M):
        L = torch.diag_embed(lmd[:, k].repeat(M, 1).transpose(0, 1))
        w = -0.5 * torch.bmm(torch.pinverse(X - L), dJdY[:, :, k].view(B, M, 1)).view(B, M)
        dJdX += torch.einsum("bi,bj->bij", w, Y[:,:,k]) + torch.einsum("bj,bi->bij", w, Y[:,:,k])
```

Recognising that X can be decomposed as $Y\Lambda Y^\mathsf{T}$ for $Y = [y_1\, y_2\, \cdots\, y_m] \in \mathbb{R}^{m\times m}$, which we obtain from the forward pass, we can write the gradient as

$$\frac{\mathrm{d}L}{\mathrm{d}X_{ij}} = \sum_{k=1}^{m} v_k^\mathsf{T} \frac{\mathrm{d}y_k}{\mathrm{d}X_{ij}} \tag{158}$$

$$= -\frac{1}{2}\sum_{k=1}^{m} v_k^\mathsf{T} Y(\Lambda - \lambda_k I)^\dagger Y^\mathsf{T}(E_{ij} + E_{ji})y_k \tag{159}$$

and significantly speed up the various pseudo-inverse calculations in the backward pass.

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

    dJdX = torch.zeros_like(X)

    # loop over eigenvalues (version 2)
    for k in range(M):
        L = lmd - lmd[:, k].view(B, 1)
        L = torch.where(torch.abs(L) < eps, 0.0, 1.0 / L).view(B, M, 1)
        pinv = torch.bmm(Y, L.view(B, M, 1) * Y.transpose(1, 2))
        w = -0.5 * torch.bmm(pinv, dJdY[:, :, k].view(B, M, 1)).view(B, M)
        dJdX += torch.einsum("bi,bj->bij", w, Y[:,:,k]) + torch.einsum("bj,bi->bij", w, Y[:,:,k])
```

A even faster implementation is possible with some rearranging of terms to reuse and vectorize calculations. Considering only the term involving E_{ij} and observing that $E_{ij}y_k = e_i e_j^\mathsf{T} y_k = Y_{jk}e_i$, we have

$$\sum_{k=1}^{m} v_k^\mathsf{T} Y(\Lambda - \lambda_k I)^\dagger Y^\mathsf{T} E_{ij} y_k = \sum_{k=1}^{m} Y_{jk} v_k^\mathsf{T} Y(\Lambda - \lambda_k I)^\dagger Y^\mathsf{T} e_i \tag{160}$$

$$= \begin{bmatrix} Y_{j1} & Y_{j2} & \cdots & Y_{jm} \end{bmatrix} \begin{bmatrix} v_1^\mathsf{T} Y(\Lambda - \lambda_1 I)^\dagger \\ v_2^\mathsf{T} Y(\Lambda - \lambda_2 I)^\dagger \\ \vdots \\ v_m^\mathsf{T} Y(\Lambda - \lambda_m I)^\dagger \end{bmatrix} Y^\mathsf{T} e_i \tag{161}$$

$$= e_j^\mathsf{T} Y \begin{bmatrix} v_1^\mathsf{T} Y(\Lambda - \lambda_1 I)^\dagger \\ v_2^\mathsf{T} Y(\Lambda - \lambda_2 I)^\dagger \\ \vdots \\ v_m^\mathsf{T} Y(\Lambda - \lambda_m I)^\dagger \end{bmatrix} Y^\mathsf{T} e_i \tag{162}$$

$$= e_j^\mathsf{T} Y \left(\begin{bmatrix} 0 & \frac{1}{\lambda_2-\lambda_1} & \cdots & \frac{1}{\lambda_m-\lambda_1} \\ \frac{1}{\lambda_1-\lambda_2} & 0 & \cdots & \frac{1}{\lambda_m-\lambda_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{\lambda_1-\lambda_m} & \frac{1}{\lambda_2-\lambda_m} & \cdots & 0 \end{bmatrix} \odot \begin{bmatrix} v_1^\mathsf{T} Y \\ v_2^\mathsf{T} Y \\ \vdots \\ v_m^\mathsf{T} Y \end{bmatrix} \right) Y^\mathsf{T} e_i \tag{163}$$

$$= e_j^\mathsf{T} Y(\tilde{\Lambda} \odot V^\mathsf{T} Y) Y^\mathsf{T} e_i \tag{164}$$

where $V = [v_1\, v_2\, \cdots\, v_m]$ and $\tilde{\Lambda}_{ij} = \frac{1}{\lambda_i - \lambda_j}$ for $i \neq j$ and zero otherwise. Equation 163 is because post-multiplying a row vector by a diagonal matrix results in scaling each element of the vector by the corresponding diagonal entry, i.e., $a^T \mathbf{diag}(b) = b^T \odot a^T$. Thus we can eliminate the explicit for-loop in our backward pass code. Note that $\tilde{\Lambda}$ is computed using PyTorch's broadcasting mechanism by subtracting $(\lambda_1, \ldots, \lambda_m)$ from its transpose and then inverting non-zero elements (Lines 7 and 8).

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

    # do all eigenvalues in one go (version 3)
    L = lmd.view(B, 1, M) - lmd.view(B, M, 1)
    L = torch.where(torch.abs(L) < eps, 0.0, 1.0 / L)
    w = torch.bmm(L * torch.bmm(dJdY.transpose(1, 2), Y), Y.transpose(1, 2))

    dJdX = torch.einsum("bjk,bki->bij", Y, w)
    dJdX = -0.5 * (dJdX + dJdX.transpose(1, 2))
```

Continuing with some further algebraic manipulation we get

$$e_j^\mathsf{T} Y(\tilde{\Lambda} \odot V^\mathsf{T} Y) Y^\mathsf{T} e_i = e_i^\mathsf{T} Y(\tilde{\Lambda}^\mathsf{T} \odot Y^\mathsf{T} V) Y^\mathsf{T} e_j \tag{165}$$

$$= \left(Y(\tilde{\Lambda}^\mathsf{T} \odot Y^\mathsf{T} V) Y^\mathsf{T} \right)_{ij} \tag{166}$$

$$= -\left(Y(\tilde{\Lambda} \odot Y^\mathsf{T} V) Y^\mathsf{T} \right)_{ij} \tag{167}$$

Evident from this result is that we can efficiently compute derivatives with respect to all components of X using a single expression, i.e.,

$$\frac{\mathrm{d}L}{\mathrm{d}X} = \frac{1}{2}\left(Y(\tilde{\Lambda} \odot Y^\mathsf{T} V) Y^\mathsf{T} \right) + \frac{1}{2}\left(Y(\tilde{\Lambda} \odot Y^\mathsf{T} V) Y^\mathsf{T} \right)^\mathsf{T} \tag{168}$$

This final expression for the backward pass calculation is implemented in the code below (Lines 11–12).

```
@staticmethod
def backward(ctx, dJdY):
    X, 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) < EigenDecompositionFcn.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
```

We evaluate the above implementation on different size problems to determine the relative speed of the forward and backward passes. The results are shown in Figure 38 (left). As expected, most of the work is performed in the forward pass (by factoring X into $Y\Lambda Y^\mathsf{T}$), with the backward pass taking roughly half the time depending on problem size.

Last we compare the speed of the different implementations of the backward pass. As with optimal transport, the implementation can make a big difference, with the naive implementation taking over 100 times longer than the final efficient version as shown in the right-hand panel of Figure 38. Given that software engineering effort and specialised machine learning knowledge are required for optimising and testing versions of the backward pass code, it is a reasonable trade-off to stop optimising the code once it runs faster (and/or is more memory efficient) than the forward pass code, at which point we start hitting diminishing returns.

4.4 Total Variation Denoising

Total variation denoising [6] is an effective technique for eliminating noise from a signal while maintaining its distinct features. The algorithm employed in total variation denoising facilitates the enhancement of a signal by removing noise while maintaining important features such as sharp signal transitions. Total variation denoising (on a one-dimensional signal) can be formulated as the following optimization problem,

$$\text{minimize}_u \quad \frac{1}{2}\|u - x\|_2^2 + \lambda \sum_{i=1}^{n-1} |u_i - u_{i+1}| \tag{169}$$

where the first term aims to match the input signal $x \in \mathbb{R}^n$ and the second term regularizes changes in the signal. Parameter $\lambda > 0$ controls the strength of regularization.

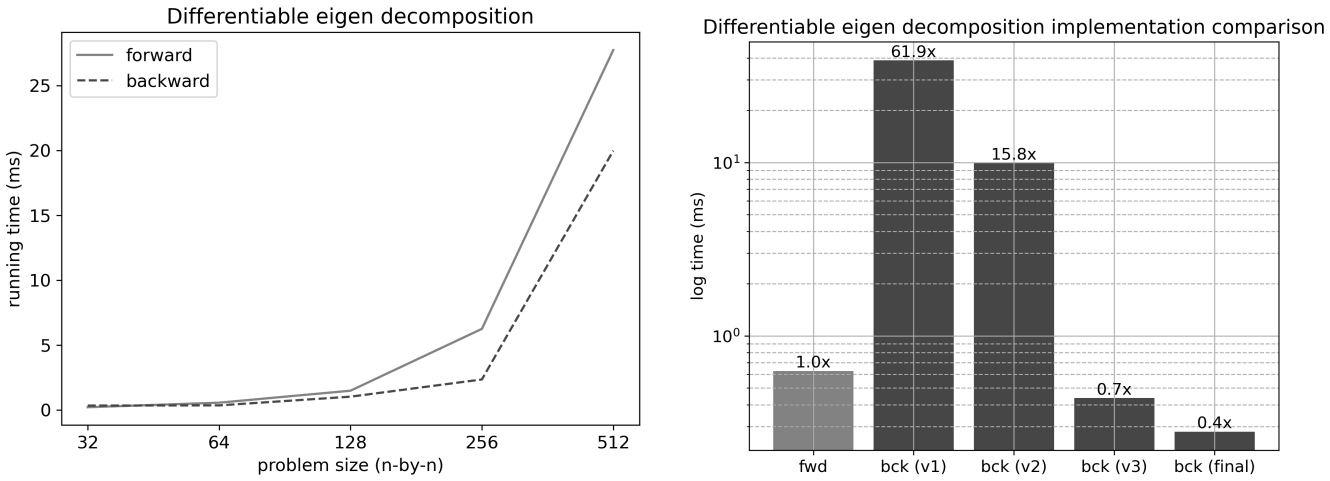


Figure 38: Left panel shows the running time of differentiable eigen decomposition for different problem sizes comparing forward and backward passes. Right panel shows the relative running time of different implementations of the backward pass for a fixed size problem (normalised against the time taken for the forward pass). All timings were averages over 1000 function calls using PyTorch 1.13.0 on a CPU.

While there are many methods that can be used to solve the total variation denoising optimization problem, we choose an iterative method based on iteratively reweighted least squares. The method computes iterates,

$$u^{(t+1)} = \left(I + \lambda D^T \Lambda_t^{-1} D\right)^{-1} x \tag{170}$$

or

$$u^{(t+1)} = \left(I - D^T \left(\frac{1}{\lambda}\Lambda_t + DD^T\right)^{-1} D\right) x \tag{171}$$

where the latter is from the Woodbury matrix identity. Here D is the discrete difference operator, i.e., $Du = \mathbf{vec}(u_i - u_{i+1})$, and $\Lambda_t = \mathbf{diag}(|Du^{(t)}|)$. We initialize $u^{(0)}$ to x and take $y = u^{(t)}$ as $t \to \infty$.

It is instructive to point out that DD^T is an $(n-1)$-by-$(n-1)$ tridiagonal matrix with two along the diagonal and negative one above and below the diagonal. Importantly, this means that $\frac{1}{\lambda}\Lambda_t + DD^T$ is a well-conditioned positive definite tridiagonal matrix, for which very fast tridiagonal solvers exist. However, at time of writing, no such solver exists in PyTorch so we opt to use Cholesky factorization here instead.

As with previous examples, e.g., optimal transport, we could easily back-propagate through the forward pass iterations using automatic differentiation during the training backward pass. However, by instead considering total variation denoising as a declarative node we can do much better. Indeed, the total variation denoising problem has the beautiful property that the derivative of the denoised signal with respect to the input signal is the identity matrix. That is,

$$\frac{\mathrm{d}y}{\mathrm{d}x} = I. \tag{172}$$

This somewhat surprising result is obvious in hindsight when we consider that the total variation objective is comprised of a quadratic term between x and u and piecewise linear terms in u. As such the backward pass can be implemented (exactly) as a *pass through* operation. It incurs no cost!

PyTorch source code for a differentiable one-dimensional total variation denoising function is shown below. The code operates on a batch of data (input signals) all with the same λ. Line 16 constructs $H = \frac{1}{\lambda}\Lambda_t + DD^T$, Line 18 computes $w = H^{-1}Dx$, and Lines 20–23 implement $u = x - D^T w$.

```
1  class TotalVariationFcn(torch.autograd.Function):
2      """PyTorch autograd function for total variation denoising."""
3
4      @staticmethod
5      def forward(ctx, x, lmd, maxiters=100):
6          with torch.no_grad():
7              # initialize u to x
8              b, n = x.shape
9              u = x.detach().clone()
10
```

```
            # iterate
            Dx = (x[:, :-1] - x[:, 1:]).view(b, n - 1, 1)
            ones = torch.ones((b, n - 2), dtype=x.dtype, device=x.device)
            for _ in range(maxiters):
                L = torch.abs(u[:, :-1] - u[:, 1:]) / lmd
                H = torch.diag_embed(L + 2.0) - torch.diag_embed(ones, offset=1) - \
                    torch.diag_embed(ones, offset=-1)
                w = torch.cholesky_solve(Dx, torch.linalg.cholesky(H)).view(b, n - 1)

                u = x.detach().clone()
                u[:, 1:] += w
                u[:, :-1] -= w

        return u

    @staticmethod
    def backward(ctx, dLdY):
        return dLdY, None, None
```

4.5 Blind PnP

An excellent example of differentiable optimal transport is in solving the blind perspective-n-point (PnP) problem. Other examples of applications that use differentiable optimal transport include semantic correspondence matching [21, 30]. For the blind PnP problem we are given an image taken by a camera and featureless 3D point cloud and our task is to estimate the position and orientation of the camera in the point cloud. That is, where the photograph was taken, but without prior knowledge of the 2D-to-3D correspondences. This is a fundamental problem for many computer vision and robotic applications, including augmented reality and visual localisation.

Since we don't have point correspondences the blind PnP problem is a chicken and egg problem. The standard (non-blind) PnP problem, where 2D-to-3D correspondences are known, is significantly less difficult. In fact, it has a closed-form solution for three points and, for a larger number of points, can be embedded in a RANSAC framework to remove outliers and find a robust solution. Similarly, knowing the camera pose makes finding the correspondences between 2D pixels in the camera plane and 3D points in space straightforward. Not knowing either makes the problem challenging especially given that we need to match features between modalities and robustly under different environment and lighting conditions.

Campbell et al. [7] solves the problem by proposing a network with two declarative nodes. The first is an optimal transport node that estimates matches between pixels and points. The second is a weighted PnP [14] solver that predicts the camera pose given the probabilistic matches. This is a nonconvex optimisation problem that makes use of RANSAC to find a good solution, but as such cannot be automatically differentiated.

The whole model, shown in Figure 39, is trained end-to-end so that we can learn good image and point cloud features for matching. One way of viewing the network in an AI context is that the feature generation provides perception for the second part, which performs reasoning. The features, i.e., from the perception task, are learned so as to optimise performance on the reasoning task.

Results from the model on benchmark datasets are impressive, achieving very high performance in a fraction of the time compared to previous approaches. Quantitative results and visualisations can be found in the paper, which compare the method to a globally optimal approach truncated to 30s running time. If left long enough the globally optimal approach would eventually find the true camera pose. The proposed method runs in a fraction of a second.

4.6 Optimal Control

Differentiable optimisation has also been applied to problems outside of the field of deep learning.[35] Constrained optimal control (COC) is the problem of finding control parameters (and state trajectory) for a dynamical system, such as a robotic arm, that minimise some objective function subject to constraints. In the discrete time, finite horizon setting, the problem can be formulate as

$$
\begin{aligned}
\text{minimize} \quad & J(x,u;\theta) \triangleq \textstyle\sum_{t=0}^{T-1} c_t(x_t,u_t;\theta) + c_T(x_T;\theta) \\
\text{subject to} \quad & x_0 = x_{\text{init}}, && && \text{(initial state)} \\
& x_{t+1} - f_t(x_t,u_t;\theta) = 0 && \forall t \in \{0,\dots,T-1\}, && \text{(dynamics)} \\
& g_t(x_t,u_t;\theta) \leq 0 && \forall t \in \{0,\dots,T-1\}, && \text{(path ineq. constraints)} \\
& h_t(x_t,u_t;\theta) = 0 && \forall t \in \{0,\dots,T-1\}, && \text{(path eq. constraints)} \\
& g_T(x_T;\theta) \leq 0, && && \text{(terminal ineq. constraints)} \\
& h_T(x_T;\theta) = 0. && && \text{(terminal eq. constraints)}
\end{aligned}
\tag{173}
$$

[35] Indeed, we already saw an example application from economics in the form of Stackelberg games.

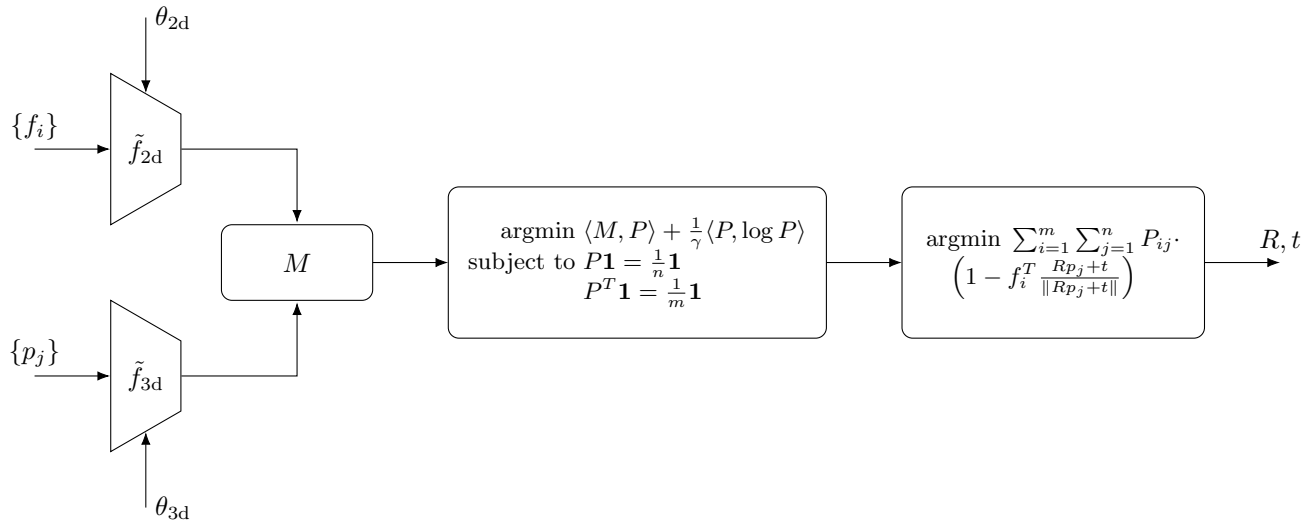


Figure 39: Network architecture for solving the blind PnP problem [7]. The network includes two declarative nodes. The first is optimal transport. The second is non-convex and employs a RANSAC algorithm to find a good solution. As such it cannot be automatically differentiated.

Here we have used standard notation from the control literature, where $x \triangleq (x_0, x_1, \ldots, x_T)$ and $u \triangleq (u_0, u_1, \ldots, u_{T-1})$ denote the state and control trajectories, respectively. The dynamics of the system at time t are governed by the current state and control input to give the next state as $x_{t+1} = f_t(x_t, u_t; \theta)$. The initial state is fixed, so the entire state trajectory is determined by the control input u. Functions g and h impose arbitrary constraints on the allowable controls and states, and functions c define the cost that we aim to minimise. All these functions can be parametrized to given a family of problems, e.g., with different coefficients of friction, system mass, etc. So the optimal solution $(x^\star, u^\star)$ can be viewed as an implicit function of the parameters θ.

It is often desirable to learn (or finetune) parameters from observed demonstration trajectories. This is known as imitation learning or learning-from-demonstration, and is essentially a bi-level optimisation problem—find parameters θ such that the optimal solution to the COC problem $(x^\star, u^\star)$ matches the demonstration trajectories as best as possible. Variants of the problem may involve considering only matching part of the COC solution, e.g., just the state trajectory and not the controls, and relate to research fields such as inverse optimal control and reinforcement learning. Solving by gradient descent requires differentiating the solution with respect to parameters. Using techniques described in the previous lecture and exploiting the structure of the COC problem, Xu et al. [38] showed that the necessary derivatives can be computed in time that is linear in the number of time steps T for the problem as well as make use of parallelism for evaluating vector-Jacobian products in the presence of general non-linear constraints. This is much faster than applying a brute force method that does not consider problem structure or ability to leverage parallel computational hardware.

4.7 Further Resources

Where to from here? We provide the following as an incomplete list of online resources with code, tutorials and examples for exploring differentiable optimisation in deep learning:

- Deep declarative networks (`http://deepdeclarativenetworks.com`)
- CVXPyLayers (`https://github.com/cvxgrp/cvxpylayers`)
- Theseus (`https://sites.google.com/view/theseus-ai`)
- JAXopt (`https://github.com/google/jaxopt`)

A Notation

Symbol	Meaning
α, β, γ	scalar constants (sometimes vectors)
θ	model parameters, linear/convex combination weights
$a, b, c \in \mathbb{R}^n$	vector constants
$e_i \in \mathbb{R}^n$, $E_{ij} \in \mathbb{R}^{m \times n}$	j-th canonical vector, (i,j)-th canonical matrix, $E_{ij} = e_i e_j^T$
$i, j, k \in \mathbb{Z}$	indices
f, g, h	functions, mappings
$m, n, p, q \in \mathbb{Z}$	dimensionality (m outputs, n inputs)
$u, v, w \in \mathbb{R}^n$	often used for temporary variables to simplify expressions
$x, y, z \in \mathbb{R}^n$	variables (multi-dimensional)
$A, B, C, H \in \mathbb{R}^{m \times n}$	matrices
$I \in \mathbb{R}^{n \times n}$	n-by-n identity matrix
$L : \mathbb{R}^n \to \mathbb{R}$	loss function (or global objective, sometimes also J); see also below
$L \in \mathbb{R}^{n \times n}$	lower triangular matrix as in LU decomposition; see also above
$Q \in \mathbb{R}^{m \times n}$	orthonormal matrix, $Q^T Q = I$
$R \in \mathbb{R}^{n \times n}$	upper triangular matrix
$\Lambda \in \mathbb{R}^{m \times m}$	diagonal matrix, $\mathbf{diag}(\lambda_1, \ldots, \lambda_m)$
$(a_1, \ldots, a_n)$	a (column) vector composed of element a_i for $i = 1, \ldots, n$, i.e., $(a_1, \ldots, a_n) = [a_1 \cdots a_n]^T$
$(\cdot)^T$	transpose of a vector or matrix
$\mathbf{1}_n$	n-dimensional vector of all ones (subscript sometime omitted)
$\mathbf{0}_n$	n-dimensional vector of all zeros (subscript sometimes omitted)
$\mathbb{R}^n, \mathbb{R}^n_+, \mathbb{R}^n_{++}$	space of n-dimensional real, non-negative, and positive vectors
$\mathbb{S}^n, \mathbb{S}^n_+, \mathbb{S}^n_{++}$	space of n-by-n symmetric, positive semi-definite and positive definite matrices
$\prec$ and $\preceq$	generalised inequalities (componentwise less than for vectors)
$\mathcal{L}(x, \lambda, \nu)$	Lagrangian (with primal variable x and dual variables λ and ν)
$\frac{\mathrm{d}f}{\mathrm{d}x}$	total derivative of scalar-valued function f with respect to scalar variable x
$\frac{\partial f}{\partial x}$	partial derivative of scalar-valued function f with respect to scalar variable x
$\nabla f \in \mathbb{R}^n$	gradient of a scalar-valued function $f : \mathbb{R}^n \to \mathbb{R}$ with respect to vector variable x
$\frac{\mathrm{d}}{\mathrm{d}x} f$ or $\mathrm{D} f \in \mathbb{R}^{m \times n}$	matrix of derivatives $\frac{\mathrm{d}f_i}{\mathrm{d}x_j}$ of a function $f : \mathbb{R}^m \to \mathbb{R}^n$
$\frac{\partial}{\partial x} f$ or $\mathrm{D}_X f \in \mathbb{R}^{m \times n}$	matrix of partial derivatives $\frac{\partial f_i}{\partial x_j}$ of a function $f : \mathbb{R}^m \times \mathbb{R}^{\tilde{m}} \to \mathbb{R}^n$
$\arg\min$	the value/index that achieves the minimum of a function/set
$\inf$	infimum, or greatest lower bound, of a set, i.e., $\inf A = \max y$ s.t. $y \leq x$ for all $x \in A$
$\mathbf{dom}\,(f)$	domain of a function (set of valid inputs)
$\mathbf{diag}(v)$	diagonal matrix formed by placing elements from vector v along the diagonal
$\mathbf{span}(v_1, \ldots, v_k)$	the space spanned by vectors $v_1, \ldots, v_k$, i.e., set of $v = \sum_{i=1}^k \theta_i v_i$ for $\theta_i \in \mathbb{R}$
$\mathbf{aff}(v_1, \ldots, v_k)$	the affine hull of vectors $v_1, \ldots, v_k$, i.e., set of $v = \sum_{i=1}^k \theta_i v_i$ for $\theta_i \in \mathbb{R}, \sum_{i=1}^k \theta_i = 1$
$\mathbf{cone}(v_1, \ldots, v_k)$	the conic hull of vectors $v_1, \ldots, v_k$, i.e., set of $v = \sum_{i=1}^k \theta_i v_i$ for $\theta_i \geq 0$
$\mathbf{conv}(v_1, \ldots, v_k)$	the convex hull of vectors $v_1, \ldots, v_k$, i.e., set of $v = \sum_{i=1}^k \theta_i v_i$ for $\theta_i \geq 0, \sum_{i=1}^k \theta_i = 1$
$\mathbf{null}(A)$	the null space of a matrix, i.e., the set $\{x \mid Ax = 0\}$
$\mathbf{rank}(A)$	the rank of a matrix, i.e., number of linearly independent rows/columns
$\mathbf{range}(A)$	the range of a matrix, i.e., space spanned by the columns of A
$\mathbf{tr}(A)$	the trace of a matrix, i.e., sum of diagonal elements of A
$[\![\cdot]\!]$	Iverson bracket. Takes value 1 if it's argument is true and 0 otherwise
$\odot, \oslash$	elementwise product and division

References

[1] B. Amos. *Differentiable Optimization-Based Modeling for Machine Learning*. PhD thesis, Carnegie Mellon University, 2019.

[2] J. F. Bard. *Practical Bilevel Optimization: Algorithms and Applications*. Kluwer Academic Press, 1998.

[3] Q. Berthet, M. Blondel, O. Teboul, M. Cuturi, J.-P. Vert, and F. Bach. Learning with differentiable perturbed optimizers. In *NeurIPS*, 2020.

[4] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 2004.

[5] M. Blondel, Q. Berthet, M. Cuturi, R. Frostig, S. Hoyer, F. Llinares-López, F. Pedregosa, and J.-P. Vert. Efficient and modular implicit differentiation. In *NeurIPS*, 2021.

[6] S. P. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge, 2004.

[7] D. Campbell, L. Liu, and S. Gould. Solving the blind perspective-n-point problem end-to-end with robust differentiable geometric optimization. In *ECCV*, 2020.

[8] M. Cuturi. Sinkhorn distances: Lightspeed computation of optimal transport. In *NeurIPS*, 2013.

[9] S. Dempe and S. Franke. On the solution of convex bilevel optimization problems. *Computational Optimization and Applications*, pages 1–19, 2015.

[10] A. L. Dontchev and R. T. Rockafellar. *Implicit Functions and Solution Mappings: A View from Variational Analysis*. Springer-Verlag, 2nd edition, 2014.

[11] S. Gould, B. Fernando, A. Cherian, P. Anderson, R. Santa Cruz, and E. Guo. On differentiating parameterized argmin and argmax problems with application to bi-level optimization. Technical report, Australian National University (arXiv:1607.05447), July 2016.

[12] S. Gould, R. Hartley, and D. Campbell. Deep declarative networks. *PAMI*, 2021.

[13] S. Gould, D. Campbell, Y. Ben-Shabat, C. H. Koneputugodage, and Z. Xu. Exploiting problem structure in deep declarative networks: Two case studies. In *First AAAI Workshop on Optimal Transport and Structured Data Modeling (OT-SDM)*, 2022.

[14] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2004.

[15] J.-B. Hiriart-Urruty and C. Lemarechal. *Fundamentals of Convex Analysis*. Springer, 2001.

[16] R. A. Horn and C. R. Johnson. *Topics in Matrix Analysis*. Cambridge University Press, 1991.

[17] S. G. Krantz and H. R. Parks. *The Implicit Function Theorem: History, Theory, and Applications*. Springer, 2013.

[18] K. Lee, S. Maji, A. Ravichandran, and S. Soatto. Meta-learning with differentiable convex optimization. In *CVPR*, 2019.

[19] D. C. Liu and J. Nocedal. On the limited memory method for large scale optimization. *Mathematical Programming B.*, 45:503–528, 1989.

[20] R. Liu, J. Gao, J. Zhang, D. Meng, and Z. Lin. Investigating bi-level optimization for learning and vision from a unified perspective: A survey and beyond. *PAMI*, 44(12), 2022.

[21] Y. Liu, L. Zhu, M. Yamada, and Y. Yang. Semantic correspondence as an optimal transport problem. In *CVPR*, 2020.

[22] J. Lorraine, P. Vicol, and D. Duvenaud. Optimizing millions of hyperparameters by implicit differentiation. In *AISTATS*, 2020.

[23] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *ICLR*, 2019.

[24] J. R. Magnus. On differentiating eigenvalues and eigenvectors. *Econometric Theory*, pages 179–191, 1985.

[25] J. R. Magnus and H. Neudecker. *Matrix Differential Calculus with Applications in Statistics and Econometrics*. John Wiley & Sons, 3rd edition, 2019.

[26] Y. Nesterov. *Introductory Lectures on Convex Optimization*. Kluwer Academic Publisher, 2004.

[27] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, 2006.

[28] H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951.

[29] R. T. Rockafellar. *Convex Analysis*. Princeton University Press, 1997.

[30] P.-E. Sarlin, D. DeTone, T. Malisiewicz, and A. Rabinovich. SuperGlue: Learning feature matching with graph neural networks. In *CVPR*, 2020.

[31] M. Spivak. *Calculus*. Cambridge University Press, 4th edition, 2008.

[32] G. Strang. *Introduction to Linear Algebra*. Cambridge University Press, 5th edition, 2016.

[33] J. Tennenbaum and B. Director. How Gauss determined the orbit of Ceres. *The American Almanac*, December 1997.

[34] M. Toso, N. D. F. Campbell, and C. Russell. Fixing implicit derivatives: Trust-region based learning of continuous energy functions. In *NeurIPS*, 2019.

[35] V. N. Vapnik. *Statistical Learning Theory*. Wiley-Interscience, 1998.

[36] M. Vlastelica, A. Paulus, V. Musil, G. Martius, and M. Rolínek. Differentiation of blackbox combinatorial solvers. In *ICLR*, 2020.

[37] H. von Stackelberg, D. Bazin, L. Urch, and R. R. Hill. *Market structure and equilibrium*. Springer, 2011.

[38] M. Xu, T. L. Molloy, and S. Gould. Revisiting implicit differentiation for learning problems in optimal control. In *NeurIPS*, 2023.