Maths for Intelligent Systems

Marc Toussaint

April, 2022

This script is primarily based on a lecture I gave 2015-2019 in Stuttgart. The current form also integrates notes and exercises from the Optimization lecture, and a little material from my Robotics and ML lectures. The first full version was from 2019, since then I occasionally update it.

Contents

1 Speaking Maths

1.1 Describing systems

Systems can be described in many ways. Biologists describe their systems often using text, and lots and lots of data. Architects describe buildings using drawings. Physicists describe nature using differential equations, or optimality principles, or differential geometry and group theory. The whole point of science is to find descriptions of systems—in the natural science descriptions that allow prediction, in the engineering sciences descriptions that enable the design of good systems, problem-solving systems.

And how should we describe intelligent systems? Robots, perception systems, machine learning systems? I think there are two main categories: the imperative way in terms of literal algorithms (code), or the declarative way in terms of formulating the problem. I prefer the latter.

The point of this lecture is to teach you to *speak maths*, to use maths to describe systems or problems. I feel that most maths courses rather teach to consume maths, or solve mathematical problems, or prove things. Clearly, this is also important. But for the purpose of intelligent systems research, it is essential to be skilled in *expressing* problems mathematically, before even thinking about solving them and deriving algorithms.

If you happen to attend a Machine Learning or Robotics course you'll see that *every* problem is addressed the same way: You have an "intuitively formulated" problem; the first step is to find a mathematical formulation; the second step to solve it. The second step is often technical. The first step is really the interesting and creative part. This is where you have to nail down the problem, i.e., nail down what it means to be successful or performant – and thereby describe "intelligence", or at least a tiny aspect of it.

The "Maths for Intelligent Systems" course will recap essentials of multi-variate functions, linear algebra, optimization, and probabilities. These fields are essential to formulate problems in intelligent systems research and hopefully will equip you with the basics

of speaking maths.

1.2 Should maths be taught with many application examples? Or abstractly?

Maybe this is the wrong question and implies a view on maths I don't agree with. I think (but this is arguable) maths is nothing but abstractions of real-world things. At least I aim to teach maths as abstractions of real-world things. It is misleading to think that there is "pure maths" and then "applications". Instead mathematical concepts, such as a vector, are abstractions of real-world things, such as faces, scenes, images, documents; and theorems, methods and algorithms that apply on vectors of course also apply to all the real-world things—subject to the limitations of this abstraction. So, the goal is not to teach you a lookup table of which method can be used in which application, but rather to teach which concepts maths offers to abstract real-world things—so that you find such abstractions yourself once you'll have to solve a real-world problem.

But yes, I believe that maths – in our context – should ideally be taught with many exercises relating to AI problems. Perhaps the ideal would be:

- Teach Maths using AI exercises (where AI problems are formulated and treated analytically).
- Teach AI using coding exercises.
- Teach coding using maths-implementation exercises.

But I haven't yet adopted this myself in my teaching.

1.3 Notation: Some seeming trivialities

Equations and mathematical expressions have a syntax. This is hardly ever made explicit[1] and might seem trivial. But it is surprising how buggy mathematical statements can be in scientific papers (and oral exams). I don't want to write much text about this, just some bullet points:

- Always declare mathematical objects.
- Be aware of variable and index scoping. For instance, if you have an equation, and one side includes a variable i, the other side doesn't, this often is a notational bug. (Unless this equation actually makes a statement about independence on i.)
- Type checking. Within an equation, be sure to know exactly of what type each term is: vector? matrix? scalar? tensor? Is the type and dimension of both sides of the equation consistent?

[1] Except perhaps by Gödel's incompleteness theorems and areas like automated theorem proving.

- Decorations are ok, but really not necessary. It is much more important to declare all things. E.g., there are all kinds of decorations used for vectors, $\boldsymbol{v}, \underline{v}, \overrightarrow{v}, |v\rangle$ and matrices. But these are not necessary. Properly declaring all symbols is much more important.

- When declaring sets of indexed elements, I use the notation $\{x_i\}_{i=1}^n$. Similarly for tuples: $(x_i)_{i=1}^n$, $(x_1, .., x_n)$, $x_{1:n}$.

- When defining sets, we write something like $\{f(x) : x \in \mathbb{R}\}$, or $\{n \in \mathbb{N} : \exists\, \{v_i\}_{i=1}^n \text{ linearly independent}, v_i \in V\}$

- I usually use brackets $[a = b] \in \{0, 1\}$ for the boolean indicator function of some expression. An alternative notation is $\mathbb{I}(a = b)$, or the **Kronecker** symbol δ_{ab}.

- A tuple $(a, b) \in A \times B$ is an element of the product space.

- **direct sum** $A \oplus B$ is same as $A \times B$ in finite-dimensional spaces

- If $f : X \to Y$, then $\min_x f(x) \in Y$ is minimal function *value* (output); whereas $\operatorname{argmin}_x f(x) \in X$ is the input ("argument") that minimizes the function. E.g., $\min_x f(x) = f(\operatorname{argmin}_x f(x))$.

- One should distinguish between the infimum $\inf_x f(x)$ and supremum $\sup_x f(x)$ from the min/max: the inf/sup refer to limits, while the min/max to values actually acquired by the function. I must admit I am sloppy in this regard and usually only write min/max.

- Never use multiple letters for one thing. E.g. $length = 3$ means l times e times n times g times t times h equals 3.

- There is a difference between $\to$ and $\mapsto$:

$$f :\ \mathbb{R} \to \mathbb{R},\ x \mapsto \cos(x) \tag{1}$$

- The dot is used to help defining functions with only some arguments fixed:

$$f :\ A \times B \to C\ , \quad f(a, \cdot) :\ B \to C,\ b \mapsto f(a, b) \tag{2}$$

  Another example is the typical declaration of an inner product: $\langle \cdot, \cdot \rangle : V \times V \to \mathbb{R}$.

- The p-**norm** or L^p-**norm** is defined as $\|x\|_p = [\sum_i x_i^p]^{1/p}$. By default $p = 2$, that is $\|x\| = \|x\|_2 = |x|$, which is the L^2-**norm** or **Euclidean length** of a vector. Also the L^1-**norm** $\|x\|_1 = \sum_i |x_i|$ (aka **Manhattan distance**) plays an important role.

- I use $\operatorname{diag}(a_1, .., a_n) = \begin{pmatrix} a_1 & & 0 \\ & \ddots & \\ 0 & & a_n \end{pmatrix}$ for a diagonal matrix. And overload this so that $\operatorname{diag}(A) = (A_{11}, .., A_{nn})$ is the diagonal vector of the matrix $A \in \mathbb{R}^{n \times n}$.

- A typical convention is

$$\mathbf{0}_n = (0, .., 0) \in \mathbb{R}^n \ , \quad \mathbf{1}_n = (1, .., 1) \in \mathbb{R}^n \ , \quad \mathbf{I}_n = \text{diag}(\mathbf{1}_n) \tag{3}$$

  Also, $e_i = (0, .., 0, 1, 0, .., 0) \in \mathbb{R}^n$ often denotes the ith column of the identity matrix, which of course are the coordinates of a basis vector $e_i \in \mathcal{V}$ in a basis $(e_i)_{i=1}^n$.

- The element-wise product of two matrices A and B is also called **Hadamard product** and notated $A \circ B$ (which has nothing to do with the concatenation of two operations). If there is need, perhaps also use this to notate the element-wise product of two vectors.

2 Functions & Derivatives

2.1 Basics on uni-variate functions

We super quickly recap the basics of functions $\mathbb{R} \to \mathbb{R}$, which the reader might already know.

2.1.1 Continuous, differentiable, & smooth functions

- A function $f : \mathbb{R} \to \mathbb{R}$ is **continuous** at x if the limit $\lim_{h \to 0} f(x + h) = f(x)$ exists and equals $f(x)$ (from both sides).

- A function $f : \mathbb{R} \to \mathbb{R}$ is **differentiable** at x if $f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$ exists. Note, differentiable $\Rightarrow$ continuous.

- A function is continuous/differentiable if it is continuous/differentiable at any $x \in \mathbb{R}$.

- A function f is an element of $\mathcal{C}^k$ if it is k-fold continuously differentiable, i.e., if its k-th derivative $f^{(k)}$ is continuous. For example, $\mathcal{C}^0$ is the space of continuous functions, and $\mathcal{C}^2$ the space of twice continuously differentiable functions.

- A function $f \in \mathcal{C}^\infty$ is called **smooth** (infinitely often differentiable).

2.1.2 Polynomials, piece-wise, basis functions, splines

Let's recap some basic functions that often appear in AI research.

- A **polynomial** of degree p is of the form $f(x) = \sum_{i=0}^{p} a_i x^i$, which is a weighed sum of **monimials** $1, x, x^2, ...$ Note that for multi-variate functions, the number of monimials grows combinatorially with the degree and dimension. E.g., the monimials of degree 2 in 3D space are x_1^2, x_1x_2, x_1x_3, x_2^2, x_2x_3, x_3^2. In general, we have $\binom{d}{p}$ monimials of degree p in d-dimensional space.

- Polynomials are smooth.

- I assume it is clear what piece-wise means (we have different polynomials in disjoint intervals covering the input domain $\mathbb{R}$).

- A set of **basis functions** $\{e_1, ..., e_n\}$ defines the space $f = \{\sum_i a_i e_i : a_i \in \mathbb{R}\}$ of functions that are linear combinations of the basis functions. More details on bases are discussed below for vector spaces in general. Here we note typical basis functions:

  - Monimials, which form the basis of polynomials.
  - Trigonometric functions $\sin(2\pi a x), \cos(2\pi a x)$, with integers a, which form the Fourier basis of functions.
  - Radial basis functions $\varphi(|x - c_i|)$, where we have n centers $c_i \in \mathbb{R}$, $i = 1, .., n$, and φ is typically some bell-shaped or local support function around zero. A very common one is the squared exponential $\varphi(d) = \exp\{-ld^2\}$ (which you might call a non-normalized Gaussian with variance $1/l$).

- The above functions are all examples of **parameteric functions**, which means that they can be specified by a finite number of parameters a_i. E.g., when we have a finite set of basis functions, the functions can all be described by the finite set of weights in the linear combination.

  However, in general a function $f : \mathbb{R} \to \mathbb{R}$ is an "infinite-dimensional object", i.e., it has infinitely many degrees-of-freedom $f(x)$, i.e., values at infinitely many points x. In fact, sometimes it is useful to think of f as a "vector" of elements f_x with continuous index x. Therefore, the space of all possible functions, and also the space of all continuous function $\mathcal{C}^0$ and smooth functions $\mathbb{C}^\infty$, is infinite-dimensional. General functions cannot be specified by a finite number of parameters, and they are called **non-parameteric**.

  The core example are functions used for regression or classification in Machine Learning, which are a linear combination of an *infinite* set of basis functions. E.g., an infinite set of radial basis functions $\varphi(|x - c|)$ for *all* centers $c \in \mathbb{R}$. This infinite set of basis functions spans a function space called Hilbert space (in the ML context, "Reproducing Kernel Hilbert Space (RKHS)"), which is an infinite-dimensional vector space. Elements in that space are called non-parameteric.

- As a final note, splines are parameteric functions that are often used in robotics and engineering in general. Splines usually are piece-wise polynomials that are continuously joined. Namely, a spline of degree p is in $\mathcal{C}^{p-1}$, i.e., $p-1$-fold continuously differentiable. A spline is not fully smooth, as the p-th derivative is

discontinuous. E.g., a cubic spline has a piece-wise constant ("bang-bang") jerk (3rd derivative). Cubic splines and B-splines (which are also piece-wise polynomials) are commonly used in robotics to describe motions. In computational design and graphics, B-splines are equally common to describe surfaces (in the form of Non-uniform rational basis spline, NURBS). Appendix **??** is a brief reference for splines.

2.2 Partial vs. total derivative, and chain rules

2.2.1 Partial Derivative

A multi-variate function $f : \mathbb{R}^n \to \mathbb{R}$ can be thought of as a function of n arguments, $f(x_1, .., x_n)$.

Definition 2.1. The *partial* derivative of a function of multiple arguments $f(x_1, .., x_n)$ is the standard derivative w.r.t. only one of its arguments,

$$\frac{\partial}{\partial x_i} f(x_1, .., x_n) = \lim_{h \to 0} \frac{f(x_1, .., x_i + h, .., x_n) - f(x)}{h} \ . \tag{4}$$

2.2.2 Total derivative, computation graphs, forward and backward chain rules

Let me start with an example: We have three real-valued quantities x, g and f which depend on each other. Specifically,

$$f(x, g) = 3x + 2g \quad \text{and} \quad g(x) = 2x \ . \tag{5}$$

Question: *What is the "derivative of f w.r.t. x"?*

The correct answer is: *Which one do you mean? The partial or total?*

The partial derivative defined above really thinks of $f(x, g)$ as a function of two arguments, and does not at all care about whether there might be dependencies of these arguments. It only looks at $f(x, g)$ alone and takes the partial derivative (=derivative w.r.t. one function argument):

$$\frac{\partial}{\partial x} f(x, g) = 3 \tag{6}$$

However, if you suddenly talk about $h(x) = f(x, g(x))$ as a function of the argument x only, that's a totally different story, and

$$\frac{\partial}{\partial x} h(x) = \frac{\partial}{\partial x} 3x + 2(2x) = 7 \tag{7}$$

Bottom line, the definition of the partial derivative really depends on what you explicitly defined as the arguments of the function.

To allow for a general treatment of differentiation with dependences we need to define a very useful concept:

Definition 2.2. A **function network** or **computation graph** is a directed acyclic graph (DAG) of n quantities x_i where each quantity is a deterministic function of a set of parents $\pi(i) \subset \{1, .., n\}$, that is

$$x_i = f_i(x_{\pi(i)}) \tag{8}$$

where $x_{\pi(i)} = (x_j)_{j \in \pi(i)}$ is the tuple of parent values. This could also be called a *deterministic Bayes net*.

In a function network all values can be computed deterministically if the input values (which do have no parents) are given. Concerning differentiation, we may now ask: Assume we have a variation dx of some input value, how do all other values vary? The chain rules give the answer. It turns our **there are two chain rules** in function networks:

Identities 2.1 (Chain rule (Forward-Version))**.**

$$\frac{df}{dx} = \sum_{g \in \pi(f)} \frac{\partial f}{\partial g} \frac{dg}{dx} \qquad \text{(with } \frac{dx}{dx} \equiv 1 \text{, in case } x \in \pi(f)) \tag{9}$$

Read this as follows: "The change of f with x is the sum of changes that come from its direct dependence on $g \in \pi(f)$, each multiplied the change of g with x."

This rule defines the **total derivative** of $\frac{df}{dx}$ w.r.t. x. Note how different these two notions of derivatives are by definition: a partial derivative only looks at a function itself and takes a limit of differences w.r.t. one argument—no notion of further dependencies. The total derivative asks how, in a function network, one value changes with a change of another.

The second version of the chain rule is:

Identities 2.2 (Chain rule (Backward-Version))**.**

$$\frac{df}{dx} = \sum_{g : x \in \pi(g)} \frac{df}{dg} \frac{\partial g}{\partial x} \qquad \text{(with } \frac{df}{df} \equiv 1 \text{, in case } x \in \pi(f)) \tag{10}$$

Read this as follows: "The change of f with x is the sum of changes that arise from all changes of g which directly depend on x."

Figure 1 illustrates the fwd and bwd versions of the chain rule. The bwd version allows you to propagate back, given gradients $\frac{df}{dg}$ from top to g, one step further down, from top to x. The fwd version allows you to propagate forward, given gradients $\frac{dg}{dx}$ from g to bottom, one step further up, from f to bottom. Both versions are recursive equations.

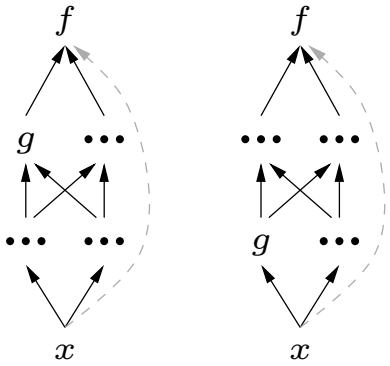


Figure 1: General Chain Rule. Left: Forward-Version, Right: Backward-Version. They gray arc denotes the direct dependence $\frac{\partial f}{\partial x}$, which appears in the summations via $dx/dx \equiv 1$, $df/df \equiv 1$.

If you would recursively plug in the definition for a given functional network, both of them would yield the same expression of $\frac{df}{dx}$ in terms of partial derivatives only.

Let's compare to the chain rule as it is commonly found in other texts (written more precisely):

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g)}{\partial g}\bigg|_{g=g(x)} \frac{\partial g(x)}{\partial x} \tag{11}$$

Note that we here very explicitly notated that $\frac{\partial f(g)}{\partial g}$ considers f to be a function of the argument g, which is evaluted at $g = g(x)$. Written like this, the rule is fine. But the above discussion and explicitly distinguishing between partial and total derivative is, when things get complicated, less prone to confusion.

2.3 Gradient, Jacobian, Hessian, Taylor Expansion

2.3.1 Gradient & Jacobian

Let's take the next step and consider functions $f : \mathbb{R}^n \to \mathbb{R}^d$ that map from n numbers to a d-dimensional output. In this case, we can take the partial derivative of each output w.r.t. each input argument, leading to a matrix of partial derivatives:

Definition 2.3. Given $f : \mathbb{R}^n \to \mathbb{R}^d$, we define the derivative (also called **Jacobian**

matrix) as

$$\frac{\partial}{\partial x} f(x) = \begin{pmatrix} \frac{\partial}{\partial x_1} f_1(x) & \frac{\partial}{\partial x_2} f_1(x) & \dots & \frac{\partial}{\partial x_n} f_1(x) \\ \frac{\partial}{\partial x_1} f_2(x) & \frac{\partial}{\partial x_2} f_2(x) & \dots & \frac{\partial}{\partial x_n} f_2(x) \\ \vdots & & & \vdots \\ \frac{\partial}{\partial x_1} f_d(x) & \frac{\partial}{\partial x_2} f_d(x) & \dots & \frac{\partial}{\partial x_n} f_d(x) \end{pmatrix} \tag{12}$$

When the function only as one output dimension, $f : \mathbb{R}^n \to \mathbb{R}^1$, the partial derivative can be written as a vector. Unlike many other texts, I advocate for consistency with the Jacobian matrix (and contra-variance, see below) and define this to be a row vector:

Definition 2.4. We define the derivative of $f : \mathbb{R}^n \to \mathbb{R}$ as the row vector of partial derivatives

$$\frac{\partial}{\partial x} f(x) = (\frac{\partial}{\partial x_1} f,\ ..,\ \frac{\partial}{\partial x_n} f) \ . \tag{13}$$

Further, we define the **gradient** as the corresponding column "vector"

$$\nabla f(x) = \Big[\frac{\partial}{\partial x} f(x)\Big]^\top \ . \tag{14}$$

The "purpose" of a derivative is to output a change of function value when being multiplied to a change of input δ. That is, in first order approximation, we have

$$f(x+\delta) - f(x) \ \doteq \ \partial f(x)\ \delta \ , \tag{15}$$

where $\doteq$ denotes "in first order approximation". This equation holds, no matter if the output space is $\mathbb{R}^d$ or $\mathbb{R}$, or the input space and variation is $\delta \in \mathbb{R}^n$ or $\delta \in \mathbb{R}$. In the gradient notation we have

$$f(x+\delta) - f(x) \ \doteq \ \nabla f(x)^\top \delta \ . \tag{16}$$

Jumping ahead to our later discussion of linear algebra: The above two equations are written in coordinates. But note that the equations are truly independent of the choice of vector space basis and independent of an optional metric or scalar product in V. The transpose should not be understood as a scalar product between two vectors, but rather as undoing the transpose in the definition of ∇f. All this is consistent to understanding the derivatives as coordinates of a 1-form, as we will introduce it later.

Given a certain direction d (with $|d| = 1$) we define the **directional derivative** as $\nabla f(x)^\top d$, and it holds

$$\nabla f(x)^\top d = \lim_{\epsilon \to 0} \frac{f(x+\epsilon d) - f(x)}{\epsilon} \ . \tag{17}$$

2.3.2 Hessian

Definition 2.5. We define the **Hessian** of a scalar function $f : \mathbb{R}^n \to \mathbb{R}$ as the symmetric matrix

$$\nabla^2 f(x) = \frac{\partial}{\partial x}\nabla f(x) = \begin{pmatrix} \frac{\partial^2}{\partial x_1 \partial x_1} f & \frac{\partial^2}{\partial x_1 \partial x_2} f & \cdots & \frac{\partial^2}{\partial x_1 \partial x_n} f \\ \frac{\partial^2}{\partial x_2 \partial x_1} f & \frac{\partial^2}{\partial x_2 \partial x_2} f & \cdots & \frac{\partial^2}{\partial x_2 \partial x_n} f \\ \vdots & & & \vdots \\ \frac{\partial^2}{\partial x_n \partial x_1} f & \frac{\partial^2}{\partial x_n \partial x_2} f & \cdots & \frac{\partial^2}{\partial x_n \partial x_n} f \end{pmatrix} \tag{18}$$

The Hessian can be thought of as the Jacobian of ∇f. Using the Hessian, we can express the 2nd order approximation of f as:

$$f(x+\delta) \;\doteq\; f(x) + \partial f(x)\ \delta + \frac{1}{2}\delta^\top\ \nabla^2 f(x)\ \delta\ . \tag{19}$$

For a uni-variate function $f : \mathbb{R} \to \mathbb{R}$, the Hessian is just a single number, namely the second derivative $f''(x)$. In this section, let's call this the "curvature" of the function (not to be confused with the Rimannian curvature of a manifold). In the uni-variate case, we have the obvious cases:

- If $f''(x) > 0$, the function is locally "curved upwared" and **convex** (see also the formal Definition **??**).
- If $f''(x) < 0$, the function is locally "curved downward" and **concave**.

In the multi-variate case $f : \mathbb{R}^n \to \mathbb{R}$, the Hessian matrix H is symmetric and we can decompose it as $H = \sum_i \lambda_i h_i h_i^\top$ with eigenvalues λ_i and eigenvectors h_i (which we will learn about in detail later). Importantly, all h_i will be orthogonal to each other, forming a nice orthonormal basis.

This insight gives us a very strong intuition on **how the Hessian H describes the local curvature** of the function f: λ_i gives the **directional curvature**, i.e., the curvature in the direction of eigenvector h_i. If $\lambda_i > 0$, f is curved upward along h_i; if $\lambda_i < 0$, f is curved downward along h_i. Therefore, the eigenvalues λ_i tell us whether the function is locally curved upward, downward, or flat in each of the orthogonal directions h_i.

This becomes particular intuitive if $\frac{\partial}{\partial x} f = 0$ is zero, i.e., the derivative (slope) of the function is zero in all directions. When the curvatures λ_i are positive in all directions, the function is locally convex (upward parabolic) and x is a local minimum; if the curvatures λ_i are all negative, the function is concave (downward parabolic) and x is a local maximum; if some curvatures are positive and some are negative along different directions h_i, then the function curves down in some directions, and up in others, and x is a **saddle point**.

Again jumping ahead, in the coordinate-free notation, the second derivative would be

defined as the 2-form

$$d^2 f\big|_x : \; V \times V \to G, \tag{20}$$

$$(v, w) \mapsto \lim_{h \to 0} \frac{df\big|_{x+hw}(v) - f\big|_x(v)}{h} \tag{21}$$

$$= \lim_{h,l \to 0} \frac{f(x + hw + lv) - f(x + hw) - f(x + lv) + f(x)}{hl} \; . \tag{22}$$

The Hessian matrix are the coordinates of this 2-form (which would actually be a row vector of row vectors).

2.3.3 Taylor expansion

In 1D, we have

$$f(x + v) \approx f(x) + f'(x)v + \frac{1}{2} f''(x) v^2 + \cdots + \frac{1}{k!} f^{(k)}(x) v^k \tag{23}$$

For $f : \mathbb{R}^n \to \mathbb{R}$, we have

$$f(x + v) \approx f(x) + \nabla f(x)^\top v + \frac{1}{2} v^\top \nabla^2 f(x) v + \cdots \tag{24}$$

which is equivalent to

$$f(x + v) \approx f(x) + \sum_j \frac{\partial}{\partial x_j} f(x) v_j + \frac{1}{2} \sum_{jk} \frac{\partial^2}{\partial x_j \partial x_k} f(x) v_j v_k + \cdots \tag{25}$$

2.4 Derivatives with matrices

The next section will introduce linear algebra from scratch – here we first want to learn how to practically deal with derivatives in matrix expressions. We think of matrices and vectors simply as arrays of numbers $\in \mathbb{R}^{n \times m}$ and $\mathbb{R}^n$. As a warmup, try to solve the following exercises:

(i) Let X, A be arbitrary matrices, A invertible. Solve for X:

$$XA + A^\top = \mathbf{I} \tag{26}$$

(ii) Let X, A, B be arbitrary matrices, $(C - 2A^\top)$ invertible. Solve for X:

$$X^\top C = [2A(X + B)]^\top \tag{27}$$

(iii) Let $x \in \mathbb{R}^n, y \in \mathbb{R}^d, A \in \mathbb{R}^{d \times n}$. A obviously *not* invertible, but let $A^\top A$ be invertible. Solve for x:

$$(Ax - y)^\top A = \mathbf{0}_n^\top \tag{28}$$

(iv) As above, additionally $B \in \mathbb{R}^{n\times n}$, B positive-definite. Solve for x:

$$(Ax - y)^\top A + x^\top B = \mathbf{0}_n^\top \tag{29}$$

(v) A core problem in Machine Learning: For $\beta \in \mathbb{R}^d,\ y \in \mathbb{R}^n,\ X \in \mathbb{R}^{n\times d}$, compute

$$\underset{\beta}{\text{argmin}} \|y - X\beta\|^2 + \lambda\|\beta\|^2 \ . \tag{30}$$

(vi) A core problem in Robotics: For $q, q_0 \in \mathbb{R}^n,\ \phi:\ \mathbb{R}^n \to \mathbb{R}^d,\ y^* \in \mathbb{R}^d$ non-linear but smooth, compute

$$\underset{q}{\text{argmin}} \|\phi(q) - y^*\|_C^2 + \|q - q_0\|_W^2 \ . \tag{31}$$

Use a local linearization of ϕ to solve this.

For problem (v), we want to find a **minimum for a matrix expression**. We find this by setting the derivative equal to zero. Here is the solution, and below details will become clear:

$$0 = \frac{\partial}{\partial\beta}\|y - X\beta\|^2 + \lambda\|\beta\|^2 \tag{32}$$

$$= 2(y - X\beta)^\top(-X) + 2\lambda\beta^\top \tag{33}$$

$$0 = -X^\top(y - X\beta) + \lambda\beta \tag{34}$$

$$0 = -X^\top y + (X^\top X + \lambda\mathbf{I})\beta \tag{35}$$

$$\beta = -(X^\top X + \lambda\mathbf{I})^{-1}X^\top y \tag{36}$$

Line 2 uses a standard rule for the derivative (see below) and gives a row vector equation. Line 3 transposes this to become a column vector equation.

2.4.1 Derivative Rules

As 2nd order terms are very common in AI methods, this is a very useful identity to learn:

Identities 2.3.

$$\frac{\partial}{\partial x}\ f(x)^\top A g(x) = f(x)^\top A \frac{\partial}{\partial x} g(x) + g(x)^\top A^\top \frac{\partial}{\partial x} f(x) \tag{37}$$

Note that using the 'gradient column' convention this reads

$$\nabla_x\ f(x)^\top A g(x) = [\frac{\partial}{\partial x} g(x)]^\top A^\top f(x) + [\frac{\partial}{\partial x} f(x)]^\top A g(x) \tag{38}$$

which I find impossible to remember, and mixes gradients-in-columns (∇) with gradients-in-rows (the Jacobian) notation.

Special cases and variants of this identity are:

$$\frac{\partial}{\partial x}[\textit{whatever}]x = [\textit{whatever}] \ , \quad \text{if } \textit{whatever} \text{ is indep. of } x \tag{39}$$

$$\frac{\partial}{\partial x}a^\top x = a^\top \tag{40}$$

$$\frac{\partial}{\partial x}Ax = A \tag{41}$$

$$\frac{\partial}{\partial x}(Ax-b)^\top(Cx-d) = (Ax-b)^\top C + (Cx-d)^\top A \tag{42}$$

$$\frac{\partial}{\partial x}x^\top Ax = x^\top A + x^\top A^\top \tag{43}$$

$$\frac{\partial}{\partial x}\|x\| = \frac{\partial}{\partial x}(x^\top x)^{\frac{1}{2}} = \frac{1}{2}(x^\top x)^{-\frac{1}{2}}\ 2x^\top = \frac{1}{\|x\|}x^\top \tag{44}$$

$$\frac{\partial^2}{\partial x^2}(Ax+a)^\top C(Bx+b) = A^\top CB + B^\top C^\top A \tag{45}$$

Further useful identities are:

Identities 2.4 (Derivative Rules)**.**

$$\frac{\partial}{\partial\theta}|A| = |A|\ \mathrm{tr}(A^{-1}\ \frac{\partial}{\partial\theta}A) \tag{46}$$

$$\frac{\partial}{\partial\theta}A^{-1} = -A^{-1}\ (\frac{\partial}{\partial\theta}A)\ A^{-1} \tag{47}$$

$$\frac{\partial}{\partial\theta}\mathrm{tr}(A) = \sum_i \frac{\partial}{\partial\theta}A_{ii} \tag{48}$$

We can also directly take a derivative of a scalar value w.r.t. a matrix:

$$\frac{\partial}{\partial X}a^\top Xb = ab^\top \tag{49}$$

$$\frac{\partial}{\partial X}(a^\top X^\top CXb) = C^\top Xab^\top + CXba^\top \tag{50}$$

$$\frac{\partial}{\partial X}\mathrm{tr}(X) = \mathbf{I} \tag{51}$$

But if this leads to confusion I would recommend to never take a derivative w.r.t. a matrix. Instead, perhaps take the derivative w.r.t. a matrix element: $\frac{\partial}{\partial X_{ij}}a^\top Xb = a_i b_j$.

For completeness, here are the most important matrix identities (the appendix lists more):

Identities 2.5 (Matrix Identities).

$$(A^{-1} + B^{-1})^{-1} = A\ (A+B)^{-1}\ B = B\ (A+B)^{-1}\ A \tag{52}$$
$$(A^{-1} - B^{-1})^{-1} = A\ (B-A)^{-1}\ B \tag{53}$$
$$(A+UBV)^{-1} = A^{-1} - A^{-1}U(B^{-1} + VA^{-1}U)^{-1}VA^{-1} \tag{54}$$
$$(A^{-1} + B^{-1})^{-1} = A - A(B+A)^{-1}A \tag{55}$$
$$(A + J^\top BJ)^{-1}J^\top B = A^{-1}J^\top(B^{-1} + JA^{-1}J^\top)^{-1} \tag{56}$$
$$(A + J^\top BJ)^{-1}A = \mathbf{I} - (A + J^\top BJ)^{-1}J^\top BJ \tag{57}$$

(54)=Woodbury; (56,57) holds for pos def A and B. See also the matrix cookbook.

2.4.2 Example: GP regression

An example from GP regression: The log-likelihood gradient w.r.t. a kernel hyperparameter:

$$\log P(y|X,b) = -\frac{1}{2}y^\top K^{-1}y - \frac{1}{2}\log|K| - \frac{n}{2}\log 2\pi \tag{58}$$
$$\text{where} \quad K_{ij} = e^{-b(x_i - x_j)^2} + \sigma^2\delta_{ij} \tag{59}$$
$$\frac{\partial}{\partial b}y^\top K^{-1}y = y^\top(-K^{-1}(\frac{\partial}{\partial b}K)K^{-1}) = y^\top(-K^{-1}AK^{-1})\ ,$$
$$\text{with}\ A_{ij} = -(x_i - x_j)^2\ e^{-b(x_i-x_j)^2} \tag{60}$$
$$\frac{\partial}{\partial b}\log|K| = \frac{1}{|K|}\ \frac{\partial}{\partial b}|K| = \frac{1}{|K|}\ |K|\ \mathrm{tr}(K^{-1}\frac{\partial}{\partial b}K) = \mathrm{tr}(K^{-1}A) \tag{61}$$

2.4.3 Example: Logistic regression

An example from logistic regression: We have the loss gradient and want the Hessian:

$$\nabla_\beta L = X^\top(p - x) + 2\lambda\mathbf{I}\beta \tag{62}$$
$$\text{where} \quad p_i = \sigma(x_i^\top\beta)\ , \quad \sigma(z) = \frac{e^z}{1+e^z}\ , \quad \sigma'(z) = \sigma(z)\ (1-\sigma(z)) \tag{63}$$
$$\nabla_\beta^2 L = \frac{\partial}{\partial\beta}\nabla_\beta L = X^\top\frac{\partial}{\partial\beta}p + 2\lambda \tag{64}$$
$$\frac{\partial}{\partial\beta}p_i = p_i(1-p_i)\ x_i^\top \tag{65}$$
$$\frac{\partial}{\partial\beta}p = \mathrm{diag}([p_i(1-p_i)]_{i=1}^n)\ X = \mathrm{diag}(p\circ(1-p))\ X \tag{66}$$
$$\nabla_\beta^2 L = X^\top\mathrm{diag}(p\circ(1-p))\ X + 2\lambda\mathbf{I} \tag{67}$$

(Where $\circ$ is the element-wise product.)

2.5 Check your gradients numerically!

This is your typical work procedure when implementing a Machine Learning or AI'ish or Optimization kind of methods:

- You first mathematically (on paper/LaTeX) formalize the problem domain, including the objective function.
- You derive analytically (on paper) the gradients/Hessian of your objective function.
- You implement the objective function and these analytic gradient equations in Matlab/Python/C++, using linear algebra packages.
- You **test the implemented gradient equations by comparing them to a finite difference estimate of the gradients**!
- Only if that works, you put everything together, interfacing the objective & gradient equations with some optimization algorithm

Algorithm 1 Finite Difference Jacobian Check

Input: $x \in \mathbb{R}^n$, function $f : \mathbb{R}^n \to \mathbb{R}^n$, function $df : \mathbb{R}^n \to \mathbb{R}^{d\times n}$

1: initialize $\hat{J} \in \mathbb{R}^{d\times m}$, and $\epsilon = 10^{-6}$
2: **for** $i = 1 : n$ **do**
3: $\quad \hat{J}_{\cdot i} = [f(x + \epsilon \boldsymbol{e}_i) - f(x - \epsilon \boldsymbol{e}_i)]/2\epsilon$ *// assigns the ith column of $\hat{J}$*
4: **end for**
5: if $\|\hat{J} - df(x)\|_\infty < 10^{-4}$ return true; else false

Here $\boldsymbol{e}_i$ is the ith standard basis vector in $\mathbb{R}^n$.

2.6 Examples and Exercises

2.6.1 More derivatives

a) In 3D, note that $a \times b = \text{skew}(a)b = -\text{skew}(b)a$, where $\text{skew}(v)$ is the skew matrix of v. What is the gradient of $(a \times b)^2$ w.r.t. a and b?

2.6.2 Multivariate Calculus

Given tensors $y \in \mathbb{R}^{a\times\dots\times z}$ and $x \in \mathbb{R}^{\alpha\times\dots\times\omega}$ where y is a function of x, the Jacobian tensor $J = \partial_x y$ is in $\mathbb{R}^{a\times\dots\times z\times\alpha\times\dots\times\omega}$ and has coefficients

$$J_{i,j,k,\dots,l,m,n\dots} = \frac{\partial}{\partial x_{l,m,n,\dots}} y_{i,j,k,\dots}$$

(All "output" indices come before all "input" indices.)

Compute the following Jacobian tensors

(i) $\frac{\partial}{\partial x} x$, where x is a vector

(ii) $\frac{\partial}{\partial x} x^\top A x$, where A is a matrix

(iii) $\frac{\partial}{\partial A} y^\top A x$, where x and y are vectors (note, we take the derivative w.r.t. A)

(iv) $\frac{\partial}{\partial A} A x$

(v) $\frac{\partial}{\partial x} f(x)^\top A g(x)$, where f and g are vector-valued functions

2.6.3 Finite Difference Gradient Checking

The following exercises will require you to code basic functions and derivatives. You can code in your prefereed language (Matlab, NumPy, Julia, whatever).

(i) Implement the following pseudo code for empirical gradient checking in the programming language of your choice:

Input: $x \in \mathbb{R}^n$, function $f : \mathbb{R}^n \to \mathbb{R}^d$, function $df : \mathbb{R}^n \to \mathbb{R}^{d\times n}$

1: initialize $\hat{J} \in \mathbb{R}^{d\times n}$, and $\epsilon = 10^{-6}$
2: **for** $i = 1 : n$ **do**
3: $\quad \hat{J}_{\cdot i} = [f(x + \epsilon e_i) - f(x - \epsilon e_i)]/2\epsilon$ *// assigns the ith column of $\hat{J}$*
4: **end for**
5: if $\|\hat{J} - df(x)\|_\infty < 10^{-4}$ return true; else false

---

Here e_i is the ith standard basis vector in $\mathbb{R}^n$.

(ii) Test this for

- $f : x \mapsto Ax$, $df : x \mapsto A$, where you sample $x \sim$`randn(n,1)` ($\mathcal{N}(0,1)$ in Matlab) and $A \sim$`randn(m,n)`
- $f : x \mapsto x^\top x$, $df : x \mapsto 2x^\top$, where $x \sim$`randn(n,1)`

2.6.4 Backprop in a Neural Net

Consider the function

$$f : \mathbb{R}^{h_0} \to \mathbb{R}^{h_3}, \quad f(x_0) = W_2 \sigma(W_1 \sigma(W_0 x_0))$$

where $W_l \in \mathbb{R}^{h_{l+1} \times h_l}$ and $\sigma : \mathbb{R} \to \mathbb{R}$ is a differentiable activation function which is applied element-wise. We also describe the function as the computation graph:

$$x_0 \;\mapsto\; z_1 = W_0 x_0 \;\mapsto\; x_1 = \sigma(z_1) \;\mapsto\; z_2 = W_1 x_1 \;\mapsto\; x_2 = \sigma(z_2) \;\mapsto\; f = W_2 x_2$$

Derive pseudo code to efficiently compute $\frac{df}{dx_0}$. (Ideally also for deeper networks.)

2.6.5 Backprop in a Neural Net

We consider again the function

$$f:\ \mathbb{R}^{h_0} \to \mathbb{R}^{h_3}\ , \quad f(x_0) = W_2\sigma(W_1\sigma(W_0 x_0))\ ,$$

where $W_l \in \mathbb{R}^{h_{l+1}\times h_l}$ and $\sigma(z) = 1/(e^{-z}+1)$ is the sigmoid function which is applied element-wise. We established last time that

$$\frac{df}{dx_0} = \frac{\partial f}{\partial x_2}\,\frac{\partial x_2}{\partial z_2}\,\frac{\partial z_2}{\partial x_1}\,\frac{\partial x_1}{\partial z_1}\,\frac{\partial z_1}{\partial x_0}$$

with:

$$\frac{\partial x_l}{\partial z_l} = \mathrm{diag}(x_l \circ (1-x_l))\ , \quad \frac{\partial z_{l+1}}{\partial x_l} = W_l\ , \quad \frac{\partial f}{\partial x_2} = W_2$$

Note: In the following we still let f be a h_3-dimensional vector. For those that are confused with the resulting tensors, simplify to f being a single scalar output.

(i) Derive also the necessary equations to get the derivative w.r.t. the weight matrices W_l, that is the Jacobian tensor

$$\frac{df}{dW_l}$$

(ii) Write code to implement $f(x)$ and $\frac{df}{dx_0}$ and $\frac{df}{dW_l}$.

To test this, choose layer sizes $(h_0, h_1, h_2, h_3) = (2, 10, 10, 2)$, i.e., 2 input and 2 output dimensions, and hidden layers of dimension 10.

For testing, choose random inputs sampled from $x \sim$`randn(2,1)`

And choose random weight matrices $W_l \sim \frac{1}{\sqrt{h_{l+1}}}$`rand(h[l+1],h[l])`.

Check the implemented Jacobian by comparing to the finite difference approximation.

Debugging Tip: If your first try does not work right away, the typical approach to debug is to "comment out" parts of your function f and df. For instance, start with testing $f(x) = W_0 x_0$; then test $f(x) = \sigma(W_0 x_0)$; then $f(x) = W_1\sigma(W_0 x_0)$; then I'm sure all bugs are found.

(iii) Bonus: Try to train the network to become the identity mapping. In the simplest case, use "stochastic gradient descent", meaning that you sample an input, compute the gradients $w_l = \frac{d(f(x)-x)^2}{dW_l}$, and make tiny updates $W_l \leftarrow W_l - \alpha w_l$.

2.6.6 Logistic Regression Gradient & Hessian

Consider the function

$$L:\ \mathbb{R}^d \to \mathbb{R}:\ L(\beta) = -\sum_{i=1}^{n} \Big[y_i \log \sigma(x_i^\top \beta) + (1-y_i) \log[1-\sigma(x_i^\top \beta)] \Big] \ +\ \lambda \beta^\top \beta \ ,$$

where $x_i \in \mathbb{R}^d$ is the ith row of a matrix $X \in \mathbb{R}^{n \times d}$, and $y \in \{0,1\}^n$ is a vector of 0s and 1s only. Here, $\sigma(z) = 1/(e^{-z}+1)$ is the sigmoid function, with $\sigma'(z) = \sigma(z)(1-\sigma(z))$.

Derive the gradient $\frac{\partial}{\partial \beta} L(\beta)$, as well as the Hessian

$$\nabla^2 L(\beta) = \frac{\partial^2}{\partial \beta^2} L(\beta) \ .$$

3 Linear Algebra

3.1 Vector Spaces

3.1.1 Why should we care for vector spaces in intelligent systems research?

We want to describe intelligent systems. For this we describe systems, or aspects of systems, as elements of a space:

- The input space X, output space Y in ML
- The space of functions (or classifiers) $f : X \to Y$ in ML
- The space of world states S and actions A in Reinforcement Learning
- The space of policies $\pi :\ S \to A$ in RL
- The space of feedback controllers $\pi :\ x \mapsto u$ in robot control
- The configuration space Q of a robot
- The space of paths $x :\ [0,1] \to Q$ in robotics
- The space of image segmentations $s :\ I \to \{0,1\}$ in computer vision

Actually, some of these spaces are not vector spaces at all. E.g. the configuration space of a robot might have 'holes', be a manifold with complex topology, or not even that (switch dimensionality at some places). But to do computations in these spaces one always either introduces (local) parameterizations that make them a vector space,[2] or one focusses on local tangent spaces (local linearizations) of these spaces, which are vector spaces.

Perhaps the most important computation we want to do in these spaces is taking derivatives—to set them equal to zero, or do gradient descent, or Newton steps for

[2] E.g. by definition an n-dimensional manifold X is locally isomorphic to $\mathbb{R}^n$.

optimization. But taking derivatives essentially requires the input space to (locally) be a vector space.[3] So, we also need vector spaces because we need derivatives, and Linear Algebra to deal with the resulting equations.

3.1.2 What is a vector?

A **vector** is nothing but an element of a vector space.

It is in general not a column, array, or tuple of numbers. (But tuples of numbers are a special case of a vector space.)

3.1.3 What is a vector space?

Definition 3.1 (vector space)**.** A **vector space**[a] V is a space (=set) on which two operations, addition and multiplication, are defined as follows

- addition $+: \ V \times V \to V$ is an abelian group, i.e.,
  - $a, b \in V \Rightarrow a + b \in V$ (closed under $+$)
  - $a + (b + c) = (a + b) + c$ (association)
  - $a + b = b + a$ (commutation)
  - $\exists$ unique $0 \in V$ s.t. $\forall v \in V: \ 0 + v = v$ (identity)
  - $\forall v \in V : \exists$ unique $-v$ s.t. $v + (-v) = 0$ (inverse)
- multiplication $\cdot: \ \mathbb{R} \times V \to V$ fulfils, for $\alpha, \beta \in \mathbb{R}$,
  - $\alpha(\beta v) = (\alpha\beta)v$ (association)
  - $1v = v$ (identity)
  - $\alpha(v + w) = \alpha v + \alpha w$ (distribution)

[a] We only consider vector spaces over $\mathbb{R}$.

Roughly, this definition says that a vector space is "closed under linear operations", meaning that we can add and scale vectors and they remain vectors.

3.2 Vectors, dual vectors, coordinates, matrices, tensors

In this section we explain what might be obvious: that once we have a basis, we can write vectors as (column) coordinate vectors, 1-forms as (row) coordinate vectors, and linear transformations as matrices. Only the last subsection becomes more practical, refers to concrete exercises, and explains how in practise not to get confused about basis

[3] Also when the space is actually a manifold; the differential is defined as a 1-form on the local tangent.

transforms and coordinate representations in different bases. So a practically oriented reader might want to skip to the last subsection.

3.2.1 A taxonomy of linear functions

For simplicity we consider only functions involving a single vector space V. But all that is said transfers to the case when multiple vector spaces $V, W, ...$ were involved.

Definition 3.2. $f : V \to X$ **linear** $\Leftrightarrow$ $f(\alpha v + \beta w) = \alpha f(v) + \beta f(w)$, where X is any other vector space (e.g. $X = \mathbb{R}$, or $X = V \times V$).

Definition 3.3. $f : V \times V \times \cdots \times V \to X$ **multi-linear** $\Leftrightarrow$ f is linear in each input.

Many names are used for special linear functions—let's make some explicit:

- $f : V \to \mathbb{R}$, called linear functional[4], or 1-form, or dual vector.
- $f : V \to V$, called linear function, or linear transform, or vector-valued 1-form
- $f : V \times V \to \mathbb{R}$, called bilinear functional, or 2-form
- $f : V \times V \times V \to \mathbb{R}$, called 3-form (or unspecifically 'multi-linear functional')
- $f : V \times V \to V$, called vector-valued 2-form (or unspecifically 'multi-linear map')
- $f : V \times V \times V \to V \times V$, called bivector-valued 3-form
- $f : V^k \to V^m$, called m-vector-valued k-form

This gives us a simple taxonomy of linear functions based on how many vectors a function *eats*, and how many it *outputs*. To give examples, consider some space X of systems (examples above), which might itself not be a vector space. But locally, around a specific $x \in X$, its tangent V is a vector space. Then

- $f : X \to \mathbb{R}$ could be a cost function over the system space.
- The differential $df|_x : V \to \mathbb{R}$ is a 1-form, telling us how f changes when 'making a tangent step' $v \in V$.
- The 2nd derivative $d^2 f|_x : V \times V \to \mathbb{R}$ is a 2-form, telling us how $df|_x(v)$ changes when 'making a tangent step' $w \in V$.
- The inner product $\langle \cdot, \cdot \rangle : V \times V \to \mathbb{R}$ is a 2-form.

Another example:

- $f : \mathbb{R}^i \to \mathbb{R}^o$ is a neural network that maps i input signals to o output signals.
- Its derivative $df|_x : \mathbb{R}^i \to \mathbb{R}^o$ is a vector-valued 1-form, telling us how each output changes with a step $v \in \mathbb{R}^i$ in the input.
- Its 2nd derivative $d^2 f|_x : \mathbb{R}^i \times \mathbb{R}^i \to \mathbb{R}^o$ is a vector-valued 2-form.

[4] The word 'functional' instead of 'function' is especially used when V is a space of functions.

This is simply to show that vector-valued functions, 1-forms, and 2-forms are common. Instead of being a neural network, f could also be a mapping from one parameterization of a system to another, or the mapping from the joint angles of a robot to its hand position.

3.2.2 Bases and coordinates

We need to define some notions. I'm not commenting on these definitions—train yourself in reading maths...

Definition 3.4. $\text{span}(\{v_i\}_{i=1}^k) = \{\sum_i \alpha_i v_i : \alpha_i \in \mathbb{R}\}$

Definition 3.5. $\{v_i\}_{i=1}^n$ **linearly independent** $\Leftrightarrow \left[\sum_i \alpha_i v_i = 0 \Rightarrow \forall_i \alpha_i = 0\right]$

Definition 3.6. $\dim(V) = \max_n\{n \in \mathbb{N} : \exists\ \{v_i\}_{i=1}^n$ lin.indep., $v_i \in V\}$

Definition 3.7. $B = (e_i)_{i=1}^n$ is a **basis** of $V \quad \Leftrightarrow \quad \text{span}(B) = V$ and B lin.indep.

Definition 3.8. The tuple $(v_1, v_2, .., v_n) \in \mathbb{R}^n$ is called **coordinates** of $v \in V$ in the basis $(e_i)_{i=1}^n$ iff $v = \sum_i v_i e_i$

Note that $\mathbb{R}^n$ is also a vector space, and therefore coordinates $v_{1:n} \in \mathbb{R}^n$ are also vectors, but in $\mathbb{R}^n$, not V. So coordinates are vectors, but vectors in general not coordinates.

Given a basis $(e_i)_{i=1}^n$, we can describe every vector v as a linear combination $v = \sum_i v_i e_i$ of *basic elements*—the basis vectors e_i. This general idea, that "linear things" can be described as linear combinations of "basic elements" carries over also to functions. In fact, to all the types of functions we described above: 1-forms, 2-forms, bi-vector-valued k-forms, whatever. And if we describe all these als linear combinations of basic elements we automatically also introduce coordinates for these things. To get there, we first have to introduce a second type of "basic elements": 1-forms.

3.2.3 The dual vector space – and its coordinates

Definition 3.9. Given V, its **dual space** is $V^* = \{f : V \to \mathbb{R}$ linear$\}$ (the space of 1-forms). Every $v^* \in V^*$ is called 1-form or **dual vector** (sometimes also *covector*).

First, it is easy to see that V^* is also a vector space: We can add two linear functionals, $f = f_1 + f_2$, and scale them, and it remains a linear functional.

Second, given a basis $(e_i)_{i=1}^n$ of V, we define a corresponding dual basis $(\widehat{e}_i)_{i=1}^n$ of V^* simply by

$$\forall_{i,j}: \quad \widehat{e}_i(e_j) = \delta_{ij} \tag{68}$$

where $\delta_{ij} = [i = j]$ is the **Kronecker delta**. Note that

$$\forall v \in V: \quad \widehat{e}_i(v) = v_i \tag{69}$$

That is, $\widehat{e}_i$ is the 1-form that simply maps a vector to its ith coordinate. It can be shown that $(\widehat{e}_i)_{i=1}^n$ is in fact a basis of V^*. (Omitted.) That tells us a lot!

$\dim(V^*) = \dim(V)$. That is, the space of 1-forms has the same dimension as V. At this place, geometric intuition should kick in: indeed, every linear function over V could be envisioned as a "plane" over V. Such a plane can be illustrated by its iso-lines and these can be uniquely determined by their orientation and distance (same dimensionality as V itself). Also, (assuming we'd know already what a transpose or scalar product is) every 1-form must be of the form $f(v) = c^\top v$ for some $c \in V$—so every f is uniquely described by a $c \in V$. Showing that the vector space V and its dual V^* are really twins.

The dual basis $(\widehat{e}_i)_{i=1}^n$ introduces coordinates in the dual space: Every 1-form f can be described as a linear combination of basis 1-forms,

$$f = \sum_i f_i \widehat{e}_i \tag{70}$$

where the tuple $(f_1, f_2, .., f_n)$ are the coordinates of f. And

$$\text{span}(\{\widehat{e}_i\}_{i=1}^n) = V^* \ . \tag{71}$$

3.2.4 Coordinates for every linear thing: tensors

We now have the *basic elements*: the basis vectors $(e_i)_{i=1}^n$ of V, and basis 1-forms $(\widehat{e}_i)_{i=1}^n$ of V^*. From these, we can describe, for instance, any bivector-valued 3-form as a linear combination as follows:

$$f: \ V \times V \times V \to V \times V \tag{72}$$

$$f = \sum_{ijklm} f_{ijklm} \ e_i \otimes e_j \otimes \widehat{e}_k \otimes \widehat{e}_l \otimes \widehat{e}_m \tag{73}$$

The $\otimes$ is called **outer product** (or **tensor product**), and $v \otimes w \in V \times W$ if V and W are finite vector spaces. For our purposes, we may think of $v \otimes w = (v, w)$ simply as the tuple of both. Therefore $e_i \otimes e_j \otimes \widehat{e}_k \otimes \widehat{e}_l \otimes \widehat{e}_m$ is a 5-tuple and we have in total n^5 such basis objects—and f_{ijklm} denotes the corresponding n^5 coordinates. The first two indices are contra-variant, the last three covariant—these notions are explained in detail later.

3.2.5 Finally: Matrices

As a special case of the above, every $f : V \to U$ can be described as a linear combination

$$f = \sum_{ij} f_{ij}\ e_i \otimes \widehat{e}_j\ , \tag{74}$$

where $(\widehat{e}_j)_{j=1}^n$ is a basis of V^* and (e_i) a basis of U.

Let's see how this fits with some easier view, without all this fuss about 1-forms. We already understood that the operator $\widehat{e}_j(v) = v_j$ simply picks the jth coordinate of a vector. Therefore

$$f(v) = \Big[\sum_{ij} f_{ij}\ e_i \otimes \widehat{e}_j\Big](v) = \sum_{ij} f_{ij}\ e_i v_j\ . \tag{75}$$

In case it helps, we can 'derive' this more slowly as

$$f(v) = f(\sum_k v_k e_k) = \sum_k v_k f(e_k) = \sum_k v_k \Big[\sum_{ij} f_{ij}\ e_i \otimes \widehat{e}_j\Big] e_k \tag{76}$$

$$= \sum_{ijk} f_{ij}\ v_k\ e_i\ \widehat{e}_j(e_k) = \sum_{ijk} f_{ij}\ v_k\ e_i\ \delta_{jk} = \sum_i \Big[\sum_j f_{ij}\ v_j\Big] e_i\ . \tag{77}$$

As a result, this tells us that the vector $u = f(v) \in V$ has the coordinates $u_i = \sum_j f_{ij}\ v_j$. And the vector $f(e_j) \in V$ has the coordinates f_{ij}, that is, $f(e_j) = \sum_i f_{ij} e_i$.

So there are n^2 coordinates f_{ij} for a linear function f. The first index is contra-variant, the second covariant (explained later). As it so happens, the whole world has agreed on a convention on how to write such coordinate numbers on sheets of 2-dimensional paper: as a **matrix**!

$$\begin{pmatrix} f_{11} & f_{12} & \cdots & f_{1n} \\ f_{21} & f_{22} & \cdots & f_{2n} \\ \vdots & & & \vdots \\ f_{n1} & f_{n2} & \cdots & f_{nn} \end{pmatrix} \tag{78}$$

The first (contra-variant) index spans columns; the second (covariant) spans rows. We call this and the respective definition of a matrix multiplication as the **matrix convention**.

Note that the identity map $\mathbf{I} : V \to V$ can be written as

$$\mathbf{I} = \sum_i e_i \otimes \widehat{e}_i\ , \quad \mathbf{I}_{ij} = \delta_{ij}\ . \tag{79}$$

Equally, the (contra-variant) coordinates of a vector are written as columns

$$\begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \tag{80}$$

and the (covariant) coordinates of a 1-form $h : V \to R$ as a row

$$(h_1 \quad h_2 \quad \cdots \quad h_n) \tag{81}$$

$u = f(v)$ is itself a vector, and its coordinates written as a column are

$$\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} f_{11} & f_{12} & \cdots & f_{1n} \\ f_{21} & f_{22} & \cdots & f_{2n} \\ \vdots & & & \vdots \\ f_{n1} & f_{n2} & \cdots & f_{nn} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \tag{82}$$

where this **matrix multiplication** is defined by $u_i = \sum_j f_{ij} v_j$, consistent to the above.

columns	rows
vector	1-form co-vector derivative
output space	input space
co-variant	contra-variant
contra-variant coordinates	co-variant coordinates

3.2.6 Coordinate transformations

The above was rather abstract. The exercises demonstrate representing vectors and transformations with coordinates and matrices in different input and output bases. We just summarize here:

- We have two bases $\mathcal{A} = (a_1, .., a_n)$ and $\mathcal{B} = (b_1, .., b_n)$, and the transformation T that maps each a_i to b_i, i.e., $\mathcal{B} = T\mathcal{A}$.
- Given a vector x we denote its **coordinates** in $\mathcal{A}$ by $[x]^A$ or briefly as x^A. And we denotes its coordinates in $\mathcal{B}$ as $[x]^B$ or x^B. E.g., x_i^A is the ith coordinate in basis $\mathcal{A}$.
- $[b_i]^A$ are the coordinates of the new basis vectors in the old basis. The **coordinate transformation matrix** B is given with elements $B_{ij} = [b_j]_i^A$. Note that

  $$[x]^A = B[x]^B \ , \tag{83}$$

  i.e., while the basis transform T carries old basis a_i vectors to new basis vectors b_i, the matrix B carries coordinates $[x]^B$ in the new basis to coordinates $[x]^A$ in the old basis! This is the origin of understanding that coordinates are contra-variant.
- Given a linear transform f in the vector space, we can represent it as a matrix in four ways, using basis $\mathcal{A}$ or $\mathcal{B}$ in the input and output spaces, respectively. If

$[f]^{AA} = F$ is the matrix in old coordinates (using $\mathcal{A}$ for input and output), then $[f]^{BB} = B^{-1}FB$ is its matrix in new coordinates, $[f]^{AB} = FB$ is its matrix using $\mathcal{B}$ for the input and $\mathcal{A}$ for the output space, and $[f]^{BA} = B^{-1}F$ is the matrix using $\mathcal{A}$ for input and $\mathcal{B}$ for output space.

- T itself is also a linear transform. $[T]^{AA} = B$ is its matrix in old coordinates. And the same $[T]^{BB} = B$ is also its matrix in new coordinates! $[T]^{BA} = \mathbf{I}$ is its matrix when using $\mathcal{A}$ for input and $\mathcal{B}$ for output space. And $[T]^{AB} = B^2$ is its matrix using $\mathcal{B}$ for input and $\mathcal{A}$ for output space.

3.3 Scalar product and orthonormal basis

Please note that so far we have not in any way referred to a scalar product or a transpose. All the concepts above, dual vector space, bases, coordinates, matrix-vector multiplication, are fully independent of the notion of a scalar product or transpose. Columns and rows naturally appear as coordinates of vectors and 1-forms. But now we need to introduce scalar products.

Definition 3.10. A **scalar product** (also called inner product) of V is a symmetric positive definite 2-form

$$\langle \cdot, \cdot \rangle : \ V \times V \to \mathbb{R} \ . \tag{84}$$

with $\langle v, w \rangle = \langle w, v \rangle$ and $\langle v, v \rangle > 0$ for all $v \neq 0 \in V$.

Definition 3.11. Given a scalar product, we define for every $v \in V$ its **dual** $v^* \in V^*$ as

$$v^* = \langle v, \cdot \rangle = \sum_i v_i \langle e_i, \cdot \rangle = \sum_i v_i e_i^* \ . \tag{85}$$

Note that $\widehat{e}_i$ and e_i^* are in general different 1-forms! The canonical dual basis $(\widehat{e}_i)_{i=1}^n$ is independent of an introduction of a scalar product, they were the basis to introduce coordinates for linear functions, including matrices. And while such coordinates do depend on a choice of basis $(e_i)_{i=1}^n$, they do *not* depend on a choice of scalar product.

The 1-forms $(e_i^*)_{i=1}^n$ also form a basis for V^*, but a different one to the canonical basis, and one that depends on the notion of a scalar product. You can see this: the coordinates v_i of v^* in the basis $(e_i^*)_{i=1}^n$ are identical to the coordinates v_i of v in the basis $(e_i)_{i=1}^n$, but different to the coordinates $(v^*)_i$ of v^* in the basis $(\widehat{e}_i)_{i=1}^n$.

Definition 3.12. Given a scalar product, a set of vectors $\{v_i\}_{i=1}^n$ is called **orthonormal** iff $\langle v_i, v_j \rangle = \delta_{ij}$.

Definition 3.13. Given a scalar product and basis $(e_i)_{i=1}^n$, we define the **metric tensor** $g_{ij} = \langle e_i, e_j \rangle$, which are the coordinates of the 2-form $\langle \cdot, \cdot \rangle$, that is

$$\langle \cdot, \cdot \rangle = \sum_{ij} g_{ij} \ \widehat{e}_i \otimes \widehat{e}_j \ . \tag{86}$$

This also implies that

$$\langle v, w \rangle = \sum_{ij} v_i w_j \langle e_i, e_j \rangle = \sum_{ij} v_i w_j g_{ij} = v^\top G w \ . \tag{87}$$

Although related, do not confuse g_{ij} with the usual definition of a metric $d(\cdot, \cdot)$ in a metric space.

3.3.1 Properties of orthonormal bases

If we have an orthonormal basis $(e_i)_{i=1}^n$, many thing simplify a lot. Throughout this subsection, we assume $\{e_i\}$ orthonormal.

- The metric tensor $g_{ij} = \langle e_i, e_j \rangle = \delta_{ij}$ is the identity matrix.[5] Such a metric is also called **Euclidean**. The norm $\|\epsilon_i\| = 1$. The canonical dual basis $(\widehat{e}_i)_{i=1}^n$ and the one defined via the scalar product $(e_i^*)_{i=1}^n$ become identical, $\widehat{e}_i = e_i^* = \langle e_i, \cdot \rangle$. Consequently, v and v^* have the same coordinates $v_i = (v^*)_i$ w.r.t. $(e_i)_{i=1}^n$ and $(\widehat{e}_i)_{i=1}^n$, respectively.

- The coordinates of vectors can now easily been computed:

$$v = \sum_i v_i e_i \quad \Rightarrow \quad \langle e_i, v \rangle = \langle e_i, \sum_j v_j e_j \rangle = \sum_j \langle e_i, e_j \rangle v_j = v_i \tag{88}$$

- The coordinates of a linear transform can equally easily been computed: Given a linear transform $f : V \to U$ an *arbitrary* (e.g. non-orthonormal) input basis $(v_i)_{i=1}^n$ of V, but an orthonormal basis $(u_i)_{i=1}^n$, then

$$f = \sum_{ij} f_{ij} u_i \otimes \widehat{v}_j \quad \Rightarrow \tag{89}$$

$$\langle u_i, f v_j \rangle = \langle u_j, \sum_{kl} f_{kl} u_k \otimes \widehat{v}_l(v_j) \rangle = \sum_{kl} f_{kl} \ \langle u_j, u_k \rangle \ \widehat{v}_l(v_j)$$

$$= \sum_{kl} f_{kl} \delta_{jk} \delta_{lj} = f_{ij} \tag{90}$$

- The projection onto a basis vector is given by $e_i \langle e_i, \cdot \rangle$.

[5] Being picky, a metric is not a matrix but a twice covariant tensor (a row of rows). That's why it is correctly called metric tensor.

- The projection onto the span of several basis vectors $(e_1, .., e_k)$ is given by $\sum_{i=1}^{k} e_i \langle e_i, \cdot \rangle$.
- The identity mapping $\mathbf{I} : V \to V$ is given by $\mathbf{I} = \sum_{i=1}^{\dim(V)} e_i \langle e_i, \cdot \rangle$.
- The scalar product with an orthonormal basis is

$$\langle v, w \rangle = \sum_{ij} v_i w_j \delta_{ij} = \sum_{i} v_i w_i \tag{91}$$

  which, using matrix convention, can also be written as

$$\langle v, w \rangle = (v_1 \ v_2 \ .. \ v_n) \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} = v^\top w = \sum_{i} v_i w_i \ , \tag{92}$$

  where for the first time we introduced the **transpose** which, in the matrix convention, swaps columns to rows and rows to columns.

As a general note, a row vector "eats a vector and outputs a scalar". That is $v^\top : \ V \to \mathbb{R}$ should be thought of as a 1-form! Due to the matrix conventions, it generally is the case that "rows eat columns", that is, every row index should always be thought of as relating to a 1-form (dual vector), and every column index as relating to a vector. That is totally consistent to our definition of coordinates.

For an orthonormal basis we also have

$$v^*(w) = \langle v, w \rangle = v^\top w \ . \tag{93}$$

That is, $v^\top$ is the coordinate representation of the 1-form v^*. (Which also says, that the coordinates of the 1-form v^* in the special basis $(e_i^*)_{i=1}^n \subset V^*$ coincide with the coordinates of the vector v.)

3.4 The Structure of Transforms & Singular Value Decomposition

We focus here on linear transforms (or "linear maps") $f : V \to U$ from one vector space to another (or the same). It turns out that such transforms have a very specific and intuitive structure, which is captured by the singular value decompositon.

3.4.1 The Singular Value Decomposition Theorem

We state the following theorem:

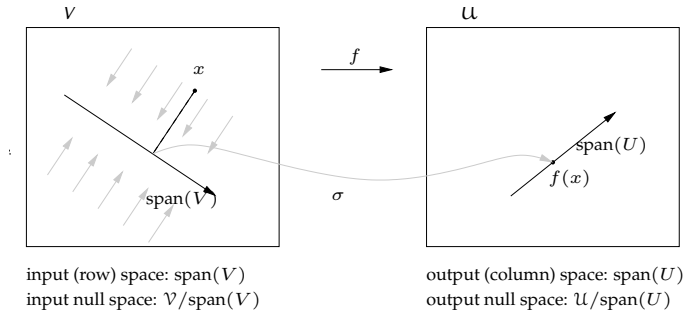


Figure 2: A linear transformation $f = \sum_{i=1}^k \sigma_i u_i v_i^\top$ can be described as: take the input x, project it onto the first input fundamental vector v_1 to yield a scalar, stretch/squeeze it by σ_1, and "unproject" this into the first output fundamental vector u_1; repeat this for all $i = 1, .., k$, and add up the results.

Theorem 3.1 (Singular Value Decomposition)**.** Given two vector spaces $\mathcal{V}$ and $\mathcal{U}$ with scalar products, $\dim(\mathcal{V}) = n$ and $\dim(\mathcal{U}) = m$, for every linear transform $f : \mathcal{V} \to \mathcal{U}$ there exist a $k \leq n, m$ and orthonormal vectors $\{v_i\}_{i=1}^k \subset \mathcal{V}$, orthonormal vectors $\{u_i\}_{i=1}^k \subset \mathcal{U}$, and positive scalars $\sigma_i > 0, i = 1, .., k$, such that

$$f = \sum_{i=1}^k \sigma_i u_i v_i^* \tag{94}$$

As above, $v_i^* = \langle v_i, \cdot \rangle$ is the basis 1-form that picks the ith coordinate of a vector in the basis $(v_i)_{i=1}^k \subset \mathcal{V}$.[a]

[a] Note that $\{v_i\}_{i=1}^k$ may not be a full basis of $\mathcal{V}$ if $k < n$. But because $\{v_i\}$ is orthonormal, $\langle v_i, \cdot \rangle$ uniquely picks the ith coordinate no matter how $\{v_i\}_{i=1}^k$ is completed with further $n - k$ vectors to become a full basis.

We first restate this theorem equivalently in coordinates.

Theorem 3.2 (Singular Value Decomposition)**.** For every matrix $A \in \mathbb{R}^{m \times n}$ there exists a $k \leq n, m$ and orthonormal vectors $\{v_i\}_{i=1}^k \subset \mathbb{R}^n$, orthonormal vectors $\{u_i\} \subset \mathbb{R}^m$, and positive scalars $\sigma_i > 0, i = 1, .., k$, such that

$$A = \sum_{i=1}^k \sigma_i u_i v_i^\top = U S V^\top \tag{95}$$

where $V = (v_1, .., v_k) \in \mathbb{R}^{n \times k}$, $U = (u_1, .., u_k) \in \mathbb{R}^{m \times k}$ contain the orthonormal bases vectors as columns and $S = \text{diag}(\sigma_1, .., \sigma_k)$.

Let me rephrase this in a sentence: *Every matrix A can be expressed as a linear combination of only k rank-1 matrices. Rank-1 matrices are the most minimalistic kinds of*

matrices and they are always of the form $uv^\top$ for some u and v. The rank-1 matrix $uv^\top$ takes an input x, projects it on v (measures its alignment with v), and "unprojects" into u (multiplies $v^\top x$ to the output vector u).

Just to explicitly show the transition from coordinate-free to the coordinate-based theorem, consider arbitrary orthonormal bases $\{e_i\}_{i=1}^n \subset \mathcal{V}$ and $\{\hat{e}_i\}_{i=1}^m \subset \mathcal{U}$. For $x \in \mathcal{V}$ we have

$$f(x) = \sum_{i=1}^k \sigma_i u_i \langle v_i, x\rangle = \sum_{i=1}^k \sigma_i (\sum_l u_{li}\hat{e}_l)\langle \sum_j v_{ji} e_j, \sum_k x_k e_k\rangle \tag{96}$$

$$= \sum_{i=1}^k \sigma_i (\sum_l u_{li}\hat{e}_l) \sum_{jk} v_{ji} x_k \delta_{jk} = \sum_l \Big[\sum_{i=1}^k u_{li}\sigma_i \sum_j v_{ji} x_j\Big]\hat{e}_l \tag{97}$$

$$= \sum_l \Big[USV^\top x\Big]_l \hat{e}_l \tag{98}$$

where v_{ji} are the coordinates of v_i, u_{li} the coordinates of u_i, $U = (u_1, .., u_k)$ is the matrix containing $\{u_i\}$ as columns, $V = (v_1, .., v_k)$ the matrix containing $\{v_i\}$ as columns, and $S = \mathrm{diag}(\sigma_1, .., \sigma_k)$ the diagonal matrix with elements σ_i.

We add some definitions based on this:

Definition 3.14. The **rank** $\mathrm{rank}(f) = \mathrm{rank}(A)$ of a transform f or its matrix A is the unique minimal k.

Definition 3.15. The **determinant** of a transform f or its matrix A is

$$\det(f) = \det(A) = \det(S) = \begin{cases} \pm \prod_{i=1}^n \sigma_i & \text{for } \mathrm{rank}(f) = n = m \\ 0 & \text{otherwise} \end{cases}, \tag{99}$$

where $\pm$ depends on whether the transform is a reflection or not.

The last definition is a bit flaky, as the $\pm$ is not properly defined. If, alternatively, in the above theorems we would require V and U to be rotations, that is, elements of $SO(n)$ (of the *special* orthogonal group); then negative σ's would indicate such a reflection and $\det(A) = \prod_{i=1}^n \sigma_i$. But above we required σ's to be strictly positive and V and U only orthogonal. Fundamental space vectors v_i and u_i could flip sign. The $\pm$ above indicates how many flip sign.

Definition 3.16. a) The **row space** (also called right or input fundamental space) of a transform f is $\mathrm{span}\{v_i\}_{i=1}^{\mathrm{rank}(f)}$. The **input null space** (or right null space) $\mathcal{V}_\perp$ is the subspace orthogonal to the row space, such that $v \in \mathcal{V}_\perp \Rightarrow f(v) = 0$.

b) The **column space** or (also called left or output fundamental space) of a transform f is $\mathrm{span}\{u_i\}_{i=1}^{\mathrm{rank}(f)}$. The **output null space** (or left null space) $\mathcal{U}_\perp$ the

subspace orthogonal to the column space, such that $u \in \mathcal{U}_\perp \Rightarrow \langle f(\cdot), u\rangle = 0$.

3.5 Point of departure from the coordinate-free notation

The coordinate-free introduction of vectors and transforms helps a lot to understand what these fundamentally are. Namely, that coordinate vectors and matrices are 'just' coordinates and rely on a choice of basis; what a metric g_{ij} really is; that only for a Euclidean metric the inner product satisfies $\langle v, w\rangle = v^\top w$. Further, the coordinate-free view is essential to understand that vector coordinates behave differently to 1-form coordinates (e.g., "gradients"!) under a transformation of the basis. We discuss contra- versus covariance of gradients at the end of this chapter.

However, we now understood that columns correspond to vectors, rows to 1-forms, and in the Euclidean case the 1-form $\langle v, \cdot\rangle$ directly corresponds to $v^\top$, in the non-Euclidean to $v^\top G$. In applications we typically represent things from start in orthonormal bases (including perhaps non-Euclidean metrics), there is not much gain sticking to the coordinate-free notation in most cases. Only when the matrix notation gets confusing (and this happens, e.g. when trying to compute something like the "Jacobian of a Jacobian", or applying the chain and product rule for a matrix expression $\partial_x f(x)^\top A(x) b(x)$) it is always a save harbour to remind what we actually talk about.

Therefore, in the rest of the notes we rely on the normal coordinate-based view. Only in some explanations we remind at the coordinate-free view when helpful.

3.6 Filling SVD with life

In the following we list some statements—all of them relate to the SVD theorem and together they're meant to give a more intuitive understanding of the equation $A = \sum_{i=1}^k \sigma_i u_i v_i^\top = U S V^\top$.

3.6.1 Understand $vv^\top$ as a projection

- The **projection** of a vector $x \in \mathcal{V}$ onto a vector $v \in \mathcal{V}$, is given by

$$x_\parallel = \frac{1}{v^2} v\langle v, x\rangle \quad \text{or} \quad \frac{vv^\top}{v^2} x \ . \tag{100}$$

  Here, the $\frac{1}{v^2}$ is normalizing in case v does not have length $|v| = 1$.

- The projection-on-v-matrix $vv^\top$ is symmetric, semi-pos-def, and has $\mathrm{rank}(vv^\top) = 1$.

- The projection of a vector $x \in \mathcal{V}$ onto a subvector space $\mathrm{span}\{v_i\}_{i=1}^k$ for *orthonor-*

mal $\{v_i\}_{i=1}^k$ is given by

$$x_{\|} = \sum_i v_i v_i^\top x = V V^T x \tag{101}$$

where $V = (v_1, .., v_k) \in \mathbb{R}^{n \times k}$. The projection matrix $VV^\top$ for orthonormal V is symmetric, semi-pos-def, and has $\text{rank}(VV^\top) = k$.

- The expression $\sum_i v_i v_i^\top$ is quite related to an SVD. Conversely it shows that the SVD represent every matrix as a liner combination of kind-of-projects, but these kind-of-projects $uv^\top$ first project onto v, but then unproject along u.

3.6.2 SVD for symmetric matrices

- Thm 2 $\Rightarrow$ Every symmetric matrix A is of the form

$$A = \sum_i \lambda_i v_i v_i^\top = V \Lambda V^\top \tag{102}$$

for orthonormal $V = (v_1, .., v_k)$. Here $\lambda_i = \pm\sigma_i$ and $\Lambda = \text{diag}(\lambda)$ is the diagonal matrix of λ's. This describes nothing but a stretching/squeezing along orthogonal projections.

- The λ_i and v_i are also the eigenvalues and eigenvectors of A, that is, for all $i = 1, .., k$:

$$A v_i = \lambda_i v_i \ . \tag{103}$$

If A has full rank, then the SVD $A = VSV^\top = VSV^{-1}$ is therefore also the eigendecomposition of A.

- The pseudo-inverse of a symmetric matrix is

$$A^\dagger = \sum_i \lambda_i^{-1} v_i v_i^\top = V S^{-1} V^\top \tag{104}$$

which simply does the reverse stretching/squeezing along the same orthogonal projections. Note that

$$A A^\dagger = A^\dagger A = V V^\top \tag{105}$$

is the projection on $\{v_i\}_{i=1}^k$. For full $\text{rank}(A) = n$ we have $VV^\top = \mathbf{I}$ and $A^\dagger = A^{-1}$. For $\text{rank}(A) < n$, we have that $A^\dagger y$ minimizes $\min_x \|Ax - y\|^2$, but there are infinitly many x's that minimize this, spanned by the null space of A. $A^\dagger y$ is the minimizer closest to zero (with smallest norm).

- Consider a data set $D = \{x_i\}_{i=1}^m$, $x_i \in \mathbb{R}^n$. For simplicity assume it has zero mean, $\sum_{i=1}^m x_i = 0$. The **covariance matrix** is defined as

$$C = \frac{1}{n} \sum_i x_i x_i^\top = \frac{1}{n} X^\top X \tag{106}$$

where (consistent to ML lecture convention) the data matrix X containes $x_i^\top$ in the ith row. Each $x_i x_i^\top$ is a projection. C is symmetric and semi-pos-dev. Using SVD we can write

$$C = \sum_i \lambda_i v_i v_i^\top \tag{107}$$

and λ_i is the data variance along the eigenvector v_i; $\sqrt{\lambda_i}$ the standard deviation along v_i; and $\sqrt{\lambda_i} v_i$ the principle axis vectors that make the ellipsoid we typically illustrate convariances with.

3.6.3 SVD for general matrices

- For full $\text{rank}(A) = n$, the determinant of a matrix is $\det(A) = \pm \prod_i \sigma_i$. We may define the **volume** spanned by any $\{b_i\}_{i=1}^n$ as

$$\text{vol}(\{b_i\}_{i=1}^n) = \det(B) \ , \quad B = (b_1, .., b_n) \in \mathbb{R}^{n \times n} \ . \tag{108}$$

It follows that

$$\text{vol}(\{Ab_i\}_{i=1}^n) = \det(A) \det(B) \tag{109}$$

that is, the volume is being multiplied with $\det(A)$, which is consistent with our intuition of transforms as stretchings/squeezings along orthonormal projections.

- The pseudo-inverse of a general matrix is

$$A^\dagger = \sum_i \sigma_i^{-1} v_i u_i^\top = V S^{-1} U^\top \ . \tag{110}$$

If $k = n$ (full input rank), $\text{rank}(A^\top A) = n$ and

$$(A^\top A)^{-1} A^\top = (V S U^\top U S V^\top)^{-1} V S U^\top = V^{-\top} S^{-2} V^{-1} V S U^\top = V S^{-1} U^\top = A^\dagger \tag{111}$$

and $A^\dagger$ is also called left pseudoinverse because $A^\dagger A = \mathbf{I}_n$.

If $k = m$ (full output rank), $\text{rank}(A A^\top) = m$ and

$$A^\top (A A^\top)^{-1} = V S U^\top (U S V^\top V S U^\top)^{-1} = V S U^\top U^{-\top} S^{-2} U^{-1} = V S^{-1} U^\top = A^\dagger \tag{112}$$

and $A^\dagger$ is also called right pseudoinverse because $A A^\dagger = \mathbf{I}_m$.

- Assume $m = n$ (same input/output dimension, or $\mathcal{V} = \mathcal{U}$), but $k < n$. Then there exist orthogonal $V, U \in \mathbb{R}^{n\times n}$ such that

$$A = UDV^\top\,, \quad D = \mathrm{diag}(\sigma_1, .., \sigma_k, 0, .., 0) = \begin{pmatrix} S & 0 \\ 0 & 0 \end{pmatrix}\,. \tag{113}$$

  Here, V and U contain a full orthonormal basis instead of only k orthonormal vectors. But the diagonal matrix D projects all but k of those to zero. Every square matrix $A \in \mathbb{R}^{n\times n}$ can be written like this.

Definition 3.17 (Rotation)**.** Given a scalar-product $\langle\cdot,\cdot\rangle$ on V, a linear transform $f : V \to V$ is called *rotation* iff it preserves the scalar product, that is,

$$\forall v, w \in V : \ \langle f(v), f(w)\rangle = \langle v, w\rangle\ . \tag{114}$$

- Every rotation matrix is orthogonal, i.e., composed of columns of orthonormal vectors.
- Every rotation has rank n and $\sigma_{1,..,n} = 1$. (No stretching/squeezing.)
- Every square matrix can be written as

$$\text{rotation}_U \cdot \text{scaling}_D \cdot \text{rotation}_{V^{-1}}$$

3.7 Eigendecomposition

Definition 3.18. The **eigendecomposition** or **diagonalization** of a square matrix $A \in \mathbb{R}^{n\times n}$ is (if it exists!)

$$A = Q\Lambda Q^{-1} \tag{115}$$

where $\Lambda = \mathrm{diag}(\lambda_1, .., \lambda_n)$ is a diagonal matrix of eigenvalues. Each column q_i of Q is an **eigenvector**.

- First note that, unlike SVD, this is not a Theorem but just a definition: If such a decomposition exists, it is called eigendecomposition. But it exists for almost any square matrix.
- The set of eigenvalues is the set of roots of the **characteristic polynomial** $p_A(\lambda) = \det(\lambda I - A)$. Why? Because then $A - \lambda I$ has 'volume' zero (or rank $< n$), showing that there exists a vector that is mapped to zero, that is $0 = (A - \lambda I)x = Ax - \lambda x$.
- Is every square matrix diagonalizable? No, but only an n^2-1-dimensional subset of matrices are not diagonalizable; most matrices are. A matrix is *not* diagonalizable if an eigenvalue λ has multiplicity k (more precisely, λ is a root of $p_A(\lambda)$ with

multiplicity k), but $n - \text{rank}(A - \lambda I)$ (the dimensionality of the span of the eigenvectors of λ!) is less than k. Therefore, the eigenvectors of λ are not linearly independent; they do not span the necessary k dimensions.

So, only very "special" matrices are not diagonalizable. Random matrices are (with prob 1).

- Symmetric matrices? $\rightarrow$ SVD
- Rotations? Not real. But complex! Think of oscillating projection onto eigenvector. If ϕ is the rotation angle, $e^{\pm i\phi}$ are eigenvalues.

3.7.1 Power Method

To find the largest eigenvector of A, initialize x randomly and iterate

$$x \leftarrow Ax \ , \quad x \leftarrow \frac{1}{\|x\|}\, x \tag{116}$$

- If this converges, x must be an eigenvector and $\lambda = x^\top A x$ the eigenvalue.
- If A is diagonalizable, and x is initially a non-zero linear combination of all eigen vectors, then it is obvious that x will converge to the "largest" (in *absolute* terms $|\lambda_i|$) eigenvector (=eigenvector with largest eigenvalue). Actually, if the largest (by norm) eigenvector is negative, then it doesn't really converge but flip sign at every iteration.

3.7.2 Power Method including the smallest eigenvalue

A trick, hard to find in the literature, to also compute the smallest eigenvalue and -vector is the following. We assume all eigenvalues to be positive. Initialize x and y randomly, iterate

$$x \leftarrow Ax \ , \quad \lambda \leftarrow \|x\| \ , \quad x \leftarrow x/\lambda \ , \quad y \leftarrow (\lambda I - A)y \ , \quad y \leftarrow y/\|y\| \tag{117}$$

Then y will converge to the smallest eigenvector, and $\lambda - \|y\|$ will be its eigenvalue. Note that (in the limit) $A - \lambda I$ only has negative eigenvalues, therefore $\|y\|$ should be positive. Finding smallest eigenvalues is a common problem in model fitting.

3.7.3 Why should I care about Eigenvalues and Eigenvectors?

- Least squares problems (finding smallest eigenvalue/-vector); (e.g. Camera Calibration, Bundle Adjustment, Plane Fitting)
- PCA
- stationary distributiuons of Markov Processes
- Page Rank
- Spectral Clustering
- Spectral Learning (e.g., as approach to training HMMs)

3.8 Beyond this script: Numerics to compute these things

We will not go into details of numerics. Nathan's script gives a really nice explaination of the QR-method. I just mention two things:

(i) The most important forms of matrices for numerics are diagonal matrices, orthogonal matrices, and upper triangular matrices. One reason is that all three types can very easily be inverted. A lot of numerics is about finding **decompositions** of general matrices into products of these special-form matrices, e.g.:

- QR-decomposition: $A = QR$ with Q orthogonal and R upper triangular.
- LU-decomposition: $A = LU$ with U and $L^\top$ upper triangular.
- Cholesky decomposition: (symmetric) $A = C^\top C$ with C upper triangular
- Eigen- & singular value decompositions

Often, these decompositions are intermediate steps to compute eigenvalue or singular value decompositions.

(ii) Use linear algebra packages. At the origin of all is LAPACK; browse through http://www.netlib.org/lapack/lug/ to get an impression of what really has been one of the most important algorithms in all technical areas of the last half century. Modern wrappers are: Matlab (Octave), which originated as just a console interface to LAPACK; the C++-library Eigen; or the Python NumPy.

3.9 Derivatives as 1-forms, steepest descent, and the covariant gradient

3.9.1 The coordinate-free view: A derivative takes a change-of-input vector as input, and returns a change of output

We previously defined the Jacobian of a function $f : \mathbb{R}^n \to \mathbb{R}^d$ from vector to vector space. We repeat defining a derivative more generally in coordinate-free form:

Definition 3.19. Given a function $f : V \to G$ we define the differential

$$df\big|_x : \; V \to G, \; v \mapsto \lim_{h \to 0} \frac{f(x + hv) - f(x)}{h} \tag{118}$$

This definition holds whenever G is a continuous space that allows the definition of this limit and the limit exists (f is differentiable). The notation $df|_x$ reads "the differential at location x", i.e., evaluating this derivative at location x.

Note that $df|_x$ is a mapping from a "tangent vector" v (a change-of-input vector) to an output-change. Further, by this definition $df|_x$ is linear. $df|_x(v)$ is the directional derivative we mentioned before. **Therefore $df|_x$ is a G-valued 1-form.** As discussed earlier, we can introduce coordinates for 1-forms; these coordinates are what typically

is called the "gradient" or "Jacobian". But here we explicitly see that we speak of coordinates of a 1-form.

3.9.2 Contra- and co-variance

In Section 3.2.6 we summarized the effects of a coorindate transformation. We recap the same here again also for derivatives and scalar products.

We have a vector space V, and a function $f : V \to \mathbb{R}$. We'll be interested in the change of function value $df|_x(\delta)$ for change of input $\delta \in V$, as well as the value of the scalar product $\langle \delta, \delta \rangle$. All these quantities are defined without any reference to coordinates; we'll check now how their coordinate representations change with a change of basis.

As in Section 3.2.6 we have two bases $\mathcal{A} = (a_1, .., a_n)$ and $\mathcal{B} = (b_1, .., b_n)$, and the transformation T that maps each a_i to b_i, i.e., $\mathcal{B} = T\mathcal{A}$. Given a vector δ we denote its coordinates in $\mathcal{A}$ by $[\delta]^A$, and its coordinates in $\mathcal{B}$ by $[\delta]^B$. Let $T^{AA} = B$ be the matrix representation of T in the old A coordinates (B contains the new basis vectors b as columns).

- We previously learned that

$$[\delta]^A = B[\delta]^B \tag{119}$$

  that is, the matrix B carries new coordinates to old ones. These coordinates are said to be contra-variant: they transform 'against' the transformation of the basis vectors.

- We require that

$$df|_x(\delta) = [\partial_x f]^A [\delta]^A = [\partial_x f]^B [\delta]^B \tag{120}$$

  must be invariant, i.e., the change of function value for δ should not depend on whether we compute it using A or B coordinates. It follows

$$[\partial_x f]^A [\delta]^A = [\partial_x f]^A B [\delta]^B = [\partial_x f]^B [\delta]^B \tag{121}$$

$$[\partial_x f]^A B = [\partial_x f]^B \tag{122}$$

  that is, the matrix B carries old 1-form-coordinates to new 1-form-coordinates. Therefore, such 1-form-coordinates are called co-variant: they transform 'with' the transformation of basis vectors.

- What we just wrote for the derivative $df|_x(\delta)$ we could equally write and argue for any 1-form $v^* \in V^*$; we always require that the value $v^*(\delta)$ is invariant.

- We also require that the scalar product $\langle \delta, \delta \rangle$ is invariant. Let

$$\langle \delta, \delta \rangle = [\delta]^{A\top} [G]^A [\delta]^A = [\delta]^{B\top} [G]^B [\delta]^B \tag{123}$$

where $[G]^A$ and $[G]^B$ are the 2-form-coordinates (metric tensor) in the old and new basis. It follows

$$[\delta]^{A\top}[G]^A[\delta]^A = [\delta]^{B\top}B^\top[G]^A B[\delta]^B \tag{124}$$

$$[G]^B = B^\top[G]^A B \tag{125}$$

that is, the matrix carries the old 2-form-coordinates to new ones. These coordinates are called twice co-variant.

Consider the following example: We have the function $f : \mathbb{R}^2 \to \mathbb{R}$, $f(x) = x_1 + x_2$. The function's partial derivative is of course $\frac{\partial f}{\partial x} = (1 \;\; 1)$. Now let's transform the coordinates of the space: we introduce new coordinates $(z_1, z_2) = (2x_1, x_2)$ or $z = B^{-1}x$ with $B = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{pmatrix}$. The same function, written in the new coordinates, is $f(z) = \frac{1}{2}z_1 + z_2$. The partial derivatives of that same function, written in these new coordinates, is $\frac{\partial f}{\partial z} = (\frac{1}{2} \;\; 1)$.

Generally, consider we have two kinds of mathematical objects and when we multiply them together this gives us a scalar. The scalar shouldn't depend on any choice of coordinate system and is therefore invariant against coordinate transforms. Then, if one of the objects transforms in a **covariant** ("transforming *with* the transformation") manner, the other object must transform in a **contra-variant** ("transforming *contrary* to the transformation") manner to ensure that the resulting scalar is invariant. This is a general principle: whenever two things multiply together to give an invariant thing, one should transform co- the other contra-variant.

Let's also check Wikipedia:

- "For a vector to be basis-independent, the components [=coordinates] of the vector must contra-vary with a change of basis to compensate. That is, the matrix that transforms the vector of components must be the inverse of the matrix that transforms the basis vectors. The components of vectors (as opposed to those of dual vectors) are said to be contravariant.
- For a dual vector (also called a covector) to be basis-independent, the components of the dual vector must co-vary with a change of basis to remain representing the same covector. That is, the components must be transformed by the same matrix as the change of basis matrix. The components of dual vectors (as opposed to those of vectors) are said to be covariant."

Ordinary gradient descent of the form $x \leftarrow x + \alpha\nabla f$ adds objects of different types: contra-variant coordinates x with co-variant partial derivatives ∇f. Clearly, adding two such different types leads to an object who's transformation under coordinate transforms is strange—and indeed the ordinary gradient descent is not invariant under transformations.

3.9.3 Steepest descent and the covariant gradient vector

Let's define the steepest descent direction to be the one where, when you make a step of length 1, you get the largest decrease of f in its linear (=1st order Taylor) approximation.

Definition 3.20. Given $f : V \to \mathbb{R}$ and a norm $\|x\|^2 = \langle x, x\rangle$ (or scalar product) defined on V, we define the **steepest descent** vector $\delta^* \in V$ as the vector:

$$\delta^* = \underset{\delta}{\text{argmin}}\, df\big|_x(\delta) \quad \text{s.t.} \quad \|\delta\|^2 = 1 \tag{126}$$

Note that for this definition we need to assume we have a scalar product, otherwise the length=1 constraint is not defined. Also recall that $df|_x(\delta) = \partial_x f(x)\delta = \nabla f(x)^\top \delta$ are equivalent notations.

Clearly, if we have coordinates in which the norm is Euclidean then

$$\|\delta\|^2 = \delta^\top\delta \quad \Rightarrow \quad \delta^* \propto -\nabla f(x) \tag{127}$$

However, if we have coordinates in which the metric is non-Euclidean, we have:

Theorem 3.3 (Steepest Descent Direction (Covariant gradient))**.** For a general scalar product $\langle v, w\rangle = v^\top G w$ (with metric tensor G), the steepest descent direction is

$$\delta^* \propto -G^{-1}\nabla f(x) \tag{128}$$

Proof: Let $G = B^\top B$ (Cholesky decomposition) and $z = B\delta$

$$\delta^* = \underset{\delta}{\text{argmin}}\, \nabla f^\top\delta \quad \text{s.t.} \quad \delta^\top G\delta = 1 \tag{129}$$

$$= B^{-1}\, \underset{z}{\text{argmin}}\, \nabla f^\top B^{-1} z \quad \text{s.t.} \quad z^\top z = 1 \tag{130}$$

$$\propto B^{-1}[-B^{-\top}\nabla f] = -G^{-1}\nabla f \tag{131}$$

For a coordinate transformation B, recall that new metric becomes $\tilde{G} = B^\top G B$, and the new gradient $\widetilde{\nabla f} = B^\top \nabla f$. Therefore, the new steepest descent is

$$\widetilde{\delta^*} = -[\tilde{G}]^{-1}\widetilde{\nabla f} = -B^{-1}G^{-1}B^{-\top}B^\top\nabla f = -B^{-1}G^{-1}\nabla f \tag{132}$$

and therefore transformes like normal contra-variant coordinates of a vector.

There is an important special case of this, when f is a function over the space of probability distributions and G is the Fisher metric, which we'll discuss later.

3.10 Examples and Exercises

3.10.1 Basis

Given a linear transform $f : \mathbb{R}^2 \to \mathbb{R}^2$,

$$f(x) = Ax = \begin{pmatrix} 7 & -10 \\ 5 & -8 \end{pmatrix} x \ .$$

Consider the basis $\mathcal{B} = \{\begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}2\\1\end{pmatrix}\}$, which we also simply refer to by the matrix $B = \begin{pmatrix}1 & 2\\1 & 1\end{pmatrix}$. Given a vector x in the vector space $\mathbb{R}^2$, we denote its coordinates in basis $\mathcal{B}$ with x^B.

(i) Show that $x = Bx^B$.

(ii) What is the matrix F^B of f in the basis $\mathcal{B}$, i.e., such that $[f(x)]^B = F^B x^B$? Prove the general equation $F^B = B^{-1}AB$.

(iii) Provide F^B numerically

Note that for a matrix $M = \begin{pmatrix}a & b\\c & d\end{pmatrix}$, $M^{-1} = \frac{1}{ad-bc}\begin{pmatrix}d & -b\\-c & a\end{pmatrix}$

3.10.2 From the Robotics Course

You have a book lying on the table. The edges of the book define the basis B, the edges of the table define basis A. Initially A and B are identical (also their origins align). Now we rotate the book by 45° counter-clock-wise about its origin.

(i) Given a dot p marked on the book at position $p^B = (1, 1)$ in the book coordinate frame, what are the coordinates p^A of that dot with respect to the table frame?

(ii) Given a point x with coordinates $x^A = (0, 1)$ in table frame, what are its coordinates x^B in the book frame?

(iii) What is the *coordinate* transformation matrix from book frame to table frame, and from table frame to book frame?

3.10.3 Bases for Polynomials

Consider the set V of all polynomials $\sum_{i=0}^{n} \alpha_i x^i$ of degree n, where $x \in \mathbb{R}$ and $\alpha_i \in \mathbb{R}$ for each $i = 0, \dots, n$.

(i) Is this set of functions a vector space? Why?

(ii) Consider two different bases of this vector space:

$$\mathcal{A} = \{1,\ x,\ x^2,\ \dots,\ x^n\}$$

and

$$\mathcal{B} = \{1,\ 1+x,\ 1+x+x^2,\ \dots,\ 1+x+\dots+x^n\}.$$

Let $f(x) = 1 + x + x^2 + x^3$ be one element in V. (This function f is a vector in the vector space V, so from here on we refer to it as a vector rather than a function.)

What are the coordinates $[f]^A$ of this vector in basis $\mathcal{A}$?

What are the coordinates $[f]^B$ of this vector in basis $\mathcal{B}$?

(iii) What matrix I^{BA} allows you to convert between coordinates $[f]^A$ and $[f]^B$, i.e. $[f]^B = I^{BA}[f]^A$? Which matrix I^{AB} does the same in the opposite direction, i.e. $[f]^A = I^{AB}[f]^B$? What is the relationship between I^{AB} and I^{BA}?

(iv) What does the difference between coordinates $[f]^A - [f]^B$ represent?

(v) Consider the linear transform $t: V \to V$ that maps basis elements of $\mathcal{A}$ directly to basis elements of $\mathcal{B}$:

$$
\begin{aligned}
&1 \to 1 \\
&x \to 1 + x \\
&x^2 \to 1 + x + x^2 \\
&\vdots \\
&x^n \to 1 + x + x^2 + \cdots + x^n
\end{aligned}
$$

- What is the matrix T^A for the linear transform t in the basis $\mathcal{A}$, i.e., such that $[t(f)]^A = T^A[f]^A$? (Basis $\mathcal{A}$ is used for both, input and output spaces.)
- What is the matrix T^B for the linear transform t in the basis $\mathcal{B}$, i.e., such that $[t(f)]^B = T^B[f]^B$? (Basis $\mathcal{B}$ is used for both, input and output spaces.)
- What is the matrix T^{BA} if we use $\mathcal{A}$ as input space basis, and $\mathcal{B}$ as output space basis, i.e., such that $[t(f)]^B = T^{BA}[f]^A$?
- What is the matrix T^{AB} if we use $\mathcal{B}$ as input space basis, and $\mathcal{A}$ as output space basis, i.e., such that $[t(f)]^A = T^{AB}[f]^B$?
- Show that $T^B = I^{BA}T^AI^{AB}$ (cp. Exercise 1(b)). Also note that $T^{AB} = T^AI^{AB}$ and $T^{BA} = I^{BA}T^A$.

3.10.4 Projections

(i) In $\mathbb{R}^n$, a plane (through the origin) is typically described by the linear equation

$$c^\top x = 0 \ , \tag{133}$$

where $c \in \mathbb{R}^n$ parameterizes the plane. Provide the matrix that describes the orthogonal projection onto this plane. (Tip: Think of the projection as $\mathbf{I}$ minus a rank-1 matrix.)

(ii) In $\mathbb{R}^n$, let's have k linearly independent $\{v_i\}_{i=1}^k$, which form the matrix $V = (v_1, .., v_k) \in \mathbb{R}^{n\times k}$. Let's formulate a projection using an optimality principle, namely,

$$\alpha^*(x) = \underset{\alpha\in\mathbb{R}^k}{\operatorname{argmin}} \|x - V\alpha\|^2 \ . \tag{134}$$

Derive the equation for the optimal $\alpha^*(x)$ from the optimality principle.

(For information only: Note that $V\alpha = \sum_{i=1}^k \alpha_i v_i$ is just the linear combination of v_i's with coefficients α. The projection of a vector x is then $x_\parallel = V\alpha^*(x)$.)

3.10.5 SVD

Consider the matrices

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \ , \quad B = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \\ 0 & 1 & 2 \end{bmatrix} \tag{135}$$

(i) Describe their 4 fundamental spaces (dimensionality, possible basis vectors).

(ii) Find the SVD of A (using pen and paper only!)

(iii) Given an arbitrary input vector $x \in \mathbb{R}^3$, provide the linear transformation matrices P_A and P_B that project x into the input null space of matrix A and B, respectively.

(iv) Compute the pseudo inverse $A^\dagger$.

(v) Determine all solutions to the linear equations $Ax = y$ and $Bx = y$ with $y = (2, 3, 0, 0)$. What is the more general expression for an arbitrary y?

3.10.6 Bonus: Scalar product and Orthogonality

(i) Show that $f(x, y) = 2x_1y_1 - x_1y_2 - x_2y_1 + 5x_2y_2$ is an scalar product on $\mathbb{R}^2$.

(ii) In the space of functions with the scalar product $\langle f, g\rangle = \int_0^{2\pi} f(x)g(x)dx$, what is the scalar product of $\sin(x)$ with $\sin(2x)$? (Graphical argument is ok.)

(iii) What property does a matrix M have to satisfy in order to be a valid metric tensor, i.e. such that $x^\top My$ is a valid scalar product?

3.10.7 Eigenvectors

(i) A symmetric matrix $A \in \mathbb{R}^{n\times n}$ is called positive semidefinite (PSD) if $x^\top Ax \geq 0, \forall x \in \mathbb{R}^n$. (PSD is usually only used with symmetric matrices.) Show that *all* eigenvalues of a PSD matrix are non-negative.

(ii) Show that if v is an eigenvector of A with eigenvalue λ, then v is also an eigenvector of A^k for any positive integer k. What is the corresponding eigenvalue?

(iii) Let v be an eigenvector of A with eigenvalue λ and w an eigenvector of $A^\top$ with a different eigenvalue $\mu \neq \lambda$. Show that v and w are orthogonal.

(iv) Suppose $A \in \mathbb{R}^{n\times n}$ has eigenvalues $\lambda_1, \dots, \lambda_n \in \mathbb{R}$. What are the eigenvalues of $A + \alpha I$ for $\alpha \in \mathbb{R}$ and I an identity matrix?

(v) Assume $A \in \mathbb{R}^{n\times n}$ is diagonalizable, i.e., it has n linearly independent eigenvectors, each with a different eigenvalue. Initialize $x \in \mathbb{R}^n$ as a random normalized vector and iterate the two steps

$$x \leftarrow Ax \ , \quad x \leftarrow \frac{1}{\|x\|}\, x$$

Prove that (under certain conditions) these iterations converge to the eigenvector x with a largest (in *absolute* terms $|\lambda_i|$) eigenvalue of A. How fast does this converge? In what sense does it converge if the largest eigenvalue is negative? What if eigenvalues are not different? Other convergence conditions?

(vi) Let A be a positive definite matrix with λ_{max} its largest eigenvalue (in absolute terms $|\lambda_i|$). What do we get when we apply power iteration method to the matrix $B = A - \lambda_{max}\mathbf{I}$? How can we get the smallest eigenvalue of A?

(vii) Consider the following variant of the previous power iteration:

$$z \leftarrow Ax \ , \quad \lambda \leftarrow x^\top z \ , \quad y \leftarrow (\lambda I - A)y \ , \quad x \leftarrow \frac{1}{\|z\|}\, z \ , \quad y \leftarrow \frac{1}{\|y\|}\, y \ .$$

If A is a positive definite matrix, show that the algorithm can give an estimate of the smallest eigenvalue of A.

3.10.8 Covariance and PCA

Suppose we're given a collection of zero-centered data points $D = \{x_i\}_{i=1}^N$, with each $x_i \in \mathbb{R}^n$. The covariance matrix is defined as

$$C = \frac{1}{n}\sum_{i=1}^{N} x_i x_i^\top = \frac{1}{n} X^\top X$$

where (consistent to ML lecture convention) the data matrix X contains each $x_i^\top$ as a row, i.e., $X^\top = (x_1, .., x_N)$.

If we project D onto some unit vector $v \in \mathbb{R}^n$, then the variance of the projected data points is $v^\top C v$. Show that the direction that maximizes this variance is the largest eigenvector of C. (Hint: Expand v in terms of the eigenvector basis of C and exploit the constraint $v^\top v = 1$.)

3.10.9 Bonus: RKHS

In machine learning we often work in spaces of functions called Reproducing Kernel Hilbert Spaces. These spaces are constructed from a certain type of function called the kernel. The kernel $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ takes two d-dimensional inputs $k(x, x')$, and from the kernel we construct a basis for the space of function, namely $B = \{k(x, \cdot)\}_{x \in \mathbb{R}^d}$. Note that this is a set of infinite element: each $x \in \mathbb{R}^d$ adds a basis function $k(x, \cdot)$ to the basis B. The scalar product between two basis functions $k_x = k(x, \cdot)$ and $k_{x'} = k(x', \cdot)$ is defined to be the kernel evaluation itself: $\langle k_x, k_{x'} \rangle = k(x, x')$. The kernel function is therefore required to be a positive definite function so that it defines a viable scalar product.

(i) Show that for any function $f \in \operatorname{span} B$ it holds

$$\langle f, k_x \rangle = f(x)$$

(ii) Assume we only have a finite set of points $\{D = \{x_i\}_{i=1}^n\}$, which defines a finite basis $\{k_{x_i}\}_{i=1}^n \subset B$. This finite function basis spans a subspace $\mathcal{F}_D = \operatorname{span}\{k_{x_i} : x_i \in D\}$ of the space of all functions.

For a general function f, we decompose it $f = f_s + f_\perp$ with $f_s \in \mathcal{F}_D$ and $\forall g \in \mathcal{F}_D : \langle f_\perp, g \rangle = 0$, i.e., $f_\perp$ is orthogonal to $\mathcal{F}_D$. Show that for every $x_i \in D$:

$$f(x_i) = f_s(x_i)$$

(Note: This shows that the function values of any function f at the data points D only depend on the part f_s which is inside the spann of $\{k_{x_i} : x_i \in D\}$. This implies the so-called representer theorem, which is fundamental in kernel machines: A loss can only depend on function values $f(x_i)$ at data points, and therefore on f_s. The part $f_\perp$ can only increase the complexity (norm) of a function. Therefore, the simplest function to optimize any loss will have $f_\perp = 0$ and be within $\operatorname{span}\{k_{x_i} : x_i \in D\}$.)

(iii) Within $\operatorname{span}\{k_{x_i} : x_i \in D\}$, what is the coordinate representation of the scalar product?

4 Optimization

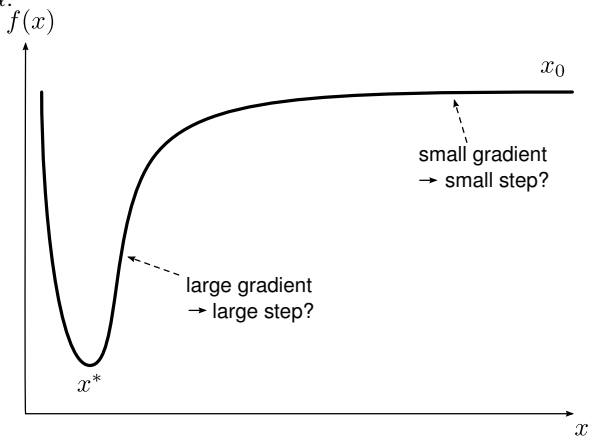
4.1 Downhill algorithms for unconstrained optimization

We discuss here algorithms that have one goal: walk downhill as quickly as possible. These target at efficiently finding local optima—in constrast to *global optimization* methods, which try to find the global optimum.

For such downhill walkers, there are two essential things to discuss: the stepsize and the step direction. When discussing the stepsize we'll hit on topics like backtracking line search, the Wolfe conditions and its implications in a basic convergence proof. The discussion of the step direction will very much circle around Newton's method and thereby also cover topics like quasi-Newton methods (BFGS), Gauss-Newton, covariant and conjugate gradients.

4.1.1 Why you shouldn't trust the magnitude of the gradient

Consider the following 1D function and naive gradient descent $x \leftarrow x - \alpha \nabla f$ for some fixed and small α.



- In plateaus we'd make small steps, at steep slopes (here close to the optimum) we make huge steps, very likely overstepping the optimum. In fact, for some α the algorithm might indefinitely loop a non-sensical sequence of very slowly walking left on the plateau, then accellerating, eventually overstepping the optimum, then being thrown back far to the right again because of the huge negative gradient on the left.

- Generally, never trust an algorithm that depends on the scaling—or choice of units—of a function! An optimization algorithm should be invariant on whether you measure costs in cents or Euros! Naive gradient descent $x \leftarrow x - \alpha \nabla f$ is not!

As a conclusion, the gradient ∇f gives a reasonable descent direction, but its magnitude is really arbitrary and no good indication of a good stepsize. Therefore, it often makes sense to just compute the step direction

$$\delta = -\frac{1}{|\nabla f(x)|}\nabla f(x) \tag{136}$$

and iterate $x \leftarrow x + \alpha\delta$ for some appropriate stepsize.

4.1.2 Ensuring monotone and sufficient decrease: Backtracking line search, Wolfe conditions, & convergence

The first idea is simple: If a step would increase the objective value, reduce the stepsize. We typically use multiplicative stepsize adaptations: Reduce $\alpha \leftarrow \varrho_\alpha^- \alpha$ with $\varrho_\alpha^- \approx 0.5$; and increase $\alpha \leftarrow \varrho_\alpha^+ \alpha$ with $\varrho_\alpha^+ \approx 1.2$. A simple monotone gradient descent algorithm reads as follows (the blue part is explained later). Here, the step vector δ is always

Algorithm 2 Plain gradient descent with backtracking line search

Input: initial $x \in \mathbb{R}^n$, functions $f(x)$ and $\nabla f(x)$, tolerance θ, parameters (defaults: $\varrho_\alpha^+ = 1.2, \varrho_\alpha^- = 0.5, \delta_{\text{max}} = \infty, \varrho_{\text{ls}} = 0.01$)

1: initialize stepsize $\alpha = 1$
2: **repeat**
3: $\quad \delta \leftarrow -\frac{\nabla f(x)}{|\nabla f(x)|}$ *// (alternative: $\delta = -\nabla f(x)$)*
4: $\quad$ **while** $f(x+\alpha\delta) > f(x) + \varrho_{\text{ls}} \nabla f(x)^\top (\alpha\delta)$ **do** *// backtracking line search*
5: $\quad\quad \alpha \leftarrow \varrho_\alpha^- \alpha$ *// decrease stepsize*
6: $\quad$ **end while**
7: $\quad x \leftarrow x + \alpha\delta$
8: $\quad \alpha \leftarrow \min\{\varrho_\alpha^+ \alpha, \delta_{\text{max}}\}$ *// increase stepsize*
9: **until** $|\alpha\delta| < \theta$ *// perhaps for 10 iterations in sequence*

normalized and α is adapted on the fly; decreasing when $f(x+\alpha\delta)$ is not sufficiently smaller than $f(x)$.

This *sufficiently smaller* is described by the blue part and is called the (1st) **Wolfe condition**

$$f(x+\alpha\delta) > f(x) + \varrho_{\text{ls}} \nabla f(x)^\top (\alpha\delta) \ . \tag{137}$$

Figure 3 illustrates this. Note that $\nabla f(x)^\top (\alpha\delta)$ is a negative value and describes how much the objective would decrease *if* f was linear. But f is of course not linear; we cannot expect that a step would really decrease f that much. Instead we require that it decreases by a fraction of this expectation. ϱ_{ls} describes this fraction and is typically chosen very moderate, e.g. $\varrho_{\text{ls}} \in [0.01, 0.1]$. So, the Wolfe conditions requires that f descreases by the ϱ_{ls}-fraction of what it would decrease if f was linear. Note that for $\alpha \to 0$ the Wolfe condition will always be fulfilled for smooth functions, because f "becomes locally linear".

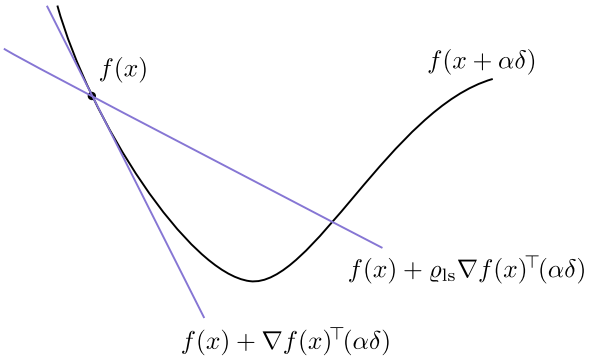


Figure 3: The 1st Wolfe condition: $f(x) + \nabla f(x)^\top (\alpha\delta)$ is the tangent, which describes the expected decrease of $f(x+\alpha\delta)$ if f *was* linear. We cannot expect this to be the case; so $f(x+\alpha\delta) > f(x) + \varrho_{\text{ls}} \nabla f(x)^\top (\alpha\delta)$ weakens this condition.

You'll proove the following theorem in the exercises. It is fundamental to *convex optimization* and proves that Alg 2 is efficient for convex objective functions:

Theorem 4.1 (Exponential convergence on convex functions)**.** Let $f : \mathbb{R}^\nu \to \mathbb{R}$ be an objective function where the eigenvalues λ of the Hessian $\nabla^2 f(x)$ are for any x. Further, assume that Hessian eigenvalues λ are lower bounded by $m > 0$ and upper bounded by $M > m$, with $m, M \in \mathbb{R}$, at any location $x \in \mathbb{R}^n$. (f is convex.) Then Algorithm 2 converges exponentially with convergence rate $(1 - 2\frac{m}{M}\varrho_{\text{ls}}\varrho_\alpha^-)$ to the optimum.

But even if your objective function is not globally convex, Alg 2 is an efficient downhill walker, and once it reaches a convex region it will efficiently walk into its local minimum.

For completeness, there is a second Wolfe condition,

$$|\nabla f(x+\alpha\delta)^\top \delta| \leq b |\nabla f(x)^\top \delta| \ , \tag{138}$$

which states that the gradient magnitude should have decreased sufficiently. We do not use it much.

4.1.3 The Newton direction

We already discussed the *steepest descent direction* $-G^{-1}\nabla f(x)$ if G is a metric tensor. Let's keep this in mind!

The original Newton method is a method to find the **root** (that is, zero point) of a function $f(x)$. In 1D it iterates $x \leftarrow x - \frac{f(x)}{f'(x)}$, that is, it uses the gradient f' to estimate where the function might cross the x-axis. To find an optimum (minimum

or maximum) of f we want to find the root of its gradient. For $x \in \mathbb{R}^n$ the Newton method iterates

$$x \leftarrow x - \nabla^2 f(x)^{-1} \nabla f(x) \ . \tag{139}$$

Note that the Newton step $\delta = -\nabla^2 f(x)^{-1} \nabla f(x)$ is the solution to

$$\min_\delta \left[f(x) + \nabla f(x)^\top \delta + \frac{1}{2} \delta^\top \nabla^2 f(x) \delta \right] \ . \tag{140}$$

So the Newton method can also be viewed as 1) compute the 2nd-order Taylor approximation to f at x, and 2) jump to the optimum of this approximation.

Note:

- If f is just a 2nd-order polynomial, the Newton method will jump to the optimum in just one step.
- *Unlike the gradient magnitude* $|\nabla f(x)|$, the magnitude of the Newton step δ is very meaningful. It is scale invariant! If you'd rescale f (trade cents by Euros), δ is unchanged. $|\delta|$ is the distance to the optimum of the 2nd-order Taylor.
- *Unlike the gradient* $\nabla f(x)$, the Newton step δ is truely a vector! The vector itself is invariant under coordinate transformations; the coordinates of δ transforms contra-variant, as it is supposed to for vector coordinates.
- **The hessian as metric, and the Newton step as steepest descent:** Assume that the hessian $H = \nabla^2 f(x)$ is pos-def. Then it fulfils all necessary conditions to define a scalar product $\langle v, w \rangle = \sum_{ij} v_i w_j H_{ij}$, where H plays the role of the metric tensor. If H was the space's metric, then the steepest descent direction is $-H^{-1} \nabla f(x)$, which is the Newton direction.

  Another way to understand the same: In the 2nd-order Taylor approximation $f(x+\delta) \approx f(x) + \nabla f(x)^\top \delta + \frac{1}{2} \delta^\top H \delta$ the Hessian plays the role of a metric tensor. Or: we may think of the function f as being an isometric parabola $f(x+\delta) \propto \langle \delta, \delta \rangle$, but we've chosen coordinates where $\langle v, v \rangle = v^\top H v$ and the parabola seems squeezed.

  Note that this discussion only holds for pos-dev hessian.

A robust Newton method is the core of many solvers, see Algorithm 3. We do backtracking line search along the Newton direction, but with maximal step size $\alpha = 1$ (the full Newton step).

We can additionally add and adapt damping to gain more robustness. Some notes on the λ:

- In Alg 3, the first line chooses λ to ensure that $(\nabla^2 f(x) + \lambda \mathbf{I})$ is indeed pos-dev—and a Newton step actually decreases f instead of seeking for a maximum. There would be other options: instead of adding to all eigenvalues we could only set the negative ones to some $\lambda > 0$.

Algorithm 3 Newton method

Input: initial $x \in \mathbb{R}^n$, functions $f(x), \nabla f(x), \nabla^2 f(x)$, tolerance θ, parameters (defaults: $\varrho_\alpha^+ = 1.2, \varrho_\alpha^- = 0.5, \varrho_\lambda^+ = \varrho_\lambda^- = 1, \varrho_{\text{ls}} = 0.01$)

1: initialize stepsize $\alpha = 1$
2: **repeat**
3: choose $\lambda > -$(minimal eigenvalue of $\nabla^2 f(x)$)
4: compute δ to solve $(\nabla^2 f(x) + \lambda \mathbf{I})\ \delta = -\nabla f(x)$
5: **while** $f(x + \alpha\delta) > f(x) + \varrho_{\text{ls}} \nabla f(x)^\top (\alpha\delta)$ **do** // *line search*
6: $\alpha \leftarrow \varrho_\alpha^- \alpha$ // *decrease stepsize*
7: optionally: $\lambda \leftarrow \varrho_\lambda^+ \lambda$ and recompute d // *increase damping*
8: **end while**
9: $x \leftarrow x + \alpha\delta$ // *step is accepted*
10: $\alpha \leftarrow \min\{\varrho_\alpha^+ \alpha, 1\}$ // *increase stepsize*
11: optionally: $\lambda \leftarrow \varrho_\lambda^- \lambda$ // *decrease damping*
12: **until** $\|\alpha\delta\|_\infty < \theta$

- δ solves the problem

$$\min_\delta \left[\nabla f(x)^\top \delta + \frac{1}{2} \delta^\top \nabla^2 f(x) \delta + \frac{1}{2} \lambda \delta^2) \right] . \tag{141}$$

  So, we added a squared potential $\lambda\delta^2$ to the local 2nd-order Taylor approximation. This is like introducing a squared penalty for large steps!

- **Trust region method:** Let's consider a different mathematical program over the step:

$$\min_\delta \nabla f(x)^\top \delta + \frac{1}{2} \delta^\top \nabla^2 f(x) \delta \ \text{ s.t. } \ \delta^2 \le \beta \tag{142}$$

  This problem wants to find the minimum of the 2nd-order Taylor (like the Newton step), but constrained to a stepsize no larger than β. This β defines the *trust region:* The region in which we trust the 2nd-order Taylor to be a reasonable enough approximation.

  Let's solve this using Lagrange parameters (as we will learn it later): Let's assume the inequality constraint is active. Then we have

$$L(\delta, \lambda) = \nabla f(x)^\top \delta + \frac{1}{2} \delta^\top \nabla^2 f(x) \delta + \lambda(\delta^2 - \beta) \tag{143}$$

$$\nabla_\delta L(\delta, \lambda) = \nabla f(x)^\top + \delta^\top (\nabla^2 f(x) + 2\lambda \mathbf{I}) \tag{144}$$

  Setting this to zero gives the step $\delta = -(\nabla^2 f(x) + 2\lambda \mathbf{I})^{-1} \nabla f(x)$.

  Therefore, the λ can be viewed as **dual variable** of a trust region method. There is no analytic relation between β and λ; we cannot determine λ directly from β. We could use a constrained optimization method, like primal-dual Newton,

or Augmented Lagrangian approach to solve for λ and δ. Alternatively, we can always increase λ when the computed steps are too large, and decrease if they are smaller than β—the Augmented Lagrangian update equations could be used to do such an update of λ.

- The $\lambda\delta^2$ term can be interpreted as penalizing the *velocity* (stepsizes) of the algorithm. This is in analogy to a damping, such as "honey pot damping", in physical dynamic systems. The parameter λ is therefore also called **damping** parameter. Such a damped (or regularized) least-squares method is also called **Levenberg-Marquardt** method.
- For $\lambda \to \infty$ the step direction δ becomes aligned with the plain gradient direction $-\nabla f(x)$. This shows that for $\lambda \to \infty$ the Hessian (and metric deformation of the space) becomes less important and instead we'll walk orthogonal to the iso-lines of the function.
- The λ term makes the δ non-scale-invariant! δ is not anymore a proper vector!

4.1.4 Least Squares & Gauss-Newton: a very important special case

A special case that appears a lot in intelligent systems is the **Least Squares** case: Consider an objective function of the form

$$f(x) = \phi(x)^\top \phi(x) = \sum_i \phi_i(x)^2 \tag{145}$$

where we call $\phi(x)$ the **cost features**. This is also called a **sum-of-squares** problem. We have

$$\nabla f(x) = 2 \frac{\partial}{\partial x} \phi(x)^\top \phi(x) \tag{146}$$

$$\nabla^2 f(x) = 2 \frac{\partial}{\partial x} \phi(x)^\top \frac{\partial}{\partial x} \phi(x) + 2\phi(x)^\top \frac{\partial^2}{\partial x^2} \phi(x) \tag{147}$$

The Gauss-Newton method is the Newton method for $f(x) = \phi(x)^\top \phi(x)$ while approximating $\nabla^2 \phi(x) \approx 0$. That is, it computes approximate Newton steps

$$\delta = -(\frac{\partial}{\partial x} \phi(x)^\top \frac{\partial}{\partial x} \phi(x) + \lambda \mathbf{I})^{-1} \frac{\partial}{\partial x} \phi(x)^\top \phi(x) \ . \tag{148}$$

Note:

- The approximate Hessian $2 \frac{\partial}{\partial x} \phi(x)^\top \frac{\partial}{\partial x} \phi(x)$ is **always semi-pos-def!** Therefore, no problems arise with negative hessian eigenvalues.
- The approximate Hessian only requires the first-order derivatives of the cost features. There is no need for computationally expensive hessians of ϕ.

Algorithm 4 Newton method – practical version realized in `rai`

Input: initial $x \in \mathbb{R}^n$, functions $f(x), \nabla f(x), \nabla^2 f(x)$
Input: stopping tolerances θ_x, θ_f
Input: parameters $\varrho_\alpha^+ = 1.2, \varrho_\alpha^- = 0.5, \lambda_0 = 0.01, \varrho_\lambda^+ = \varrho_\lambda^- = 1, \varrho_{\text{ls}} = 0.01$
Input: maximal stepsize $\delta_{\text{max}} \in \mathbb{R}$, optionally lower/upper bounds $\underline{x}, \overline{x} \in \mathbb{R}^n$

1: initialize stepsize $\alpha = 1$, $\lambda = \lambda_0$
2: **repeat**
3: compute smallest eigenvalue σ_{min} of $\nabla^2 f(x) + \lambda \mathbf{I}$
4: **if** $\sigma_{\text{min}} > 0$ is sufficiently posititive **then**
5: compute δ to solve $(\nabla^2 f(x) + \lambda \mathbf{I})\ \delta = -\nabla f(x)$
6: **else**
7: Option 1: $\lambda \leftarrow 2\lambda - \sigma_{\text{min}}$ and goto line 3 *// increase regularization*
8: Option 2: $\delta \leftarrow -\delta_{\text{max}}\ \nabla f(x)/|\nabla f(x)|$ *// gradient step of length δ_{max}*
9: **end if**
10: if $\|\delta\|_\infty > \delta_{\text{max}}$: $\delta \leftarrow (\delta_{\text{max}}/\|\delta\|_\infty)\ \delta$ *// cap δ length*
11: $y \leftarrow \textsc{BoundClip}(x + \alpha\delta, \underline{x}, \overline{x})$
12: **while** $f(y) > f(x) + \varrho_{\text{ls}} \nabla f(x)^\top (y - x)$ **do** *// bound-projected line search*
13: $\alpha \leftarrow \varrho_\alpha^- \alpha$ *// decrease stepsize*
14: (unusual option: $\lambda \leftarrow \varrho_\lambda^+ \lambda$ and goto line 3) *// increase damping*
15: $y \leftarrow \textsc{BoundClip}(x + \alpha\delta, \underline{x}, \overline{x})$
16: **end while**
17: $x_{\text{old}} \leftarrow x$
18: $x \leftarrow y$ *// step is accepted*
19: $\alpha \leftarrow \min\{\varrho_\alpha^+ \alpha, 1\}$ *// increase stepsize*
20: (unusual option: $\lambda \leftarrow \varrho_\lambda^- \lambda$) *// decrease damping*
21: **until** $\|x_{\text{old}} - x\|_\infty < \theta_x$ repeatedly, or $f(x_{\text{old}}) - f(x) < \theta_f$ repeatedly

22: **procedure** $\textsc{BoundClip}(x, \underline{x}, \overline{x})$
23: $\forall_i :\quad x_i \leftarrow \min\{x_i, \overline{x}_i\}\ ,\quad x_i \leftarrow \max\{x_i, \underline{x}_i\}$
24: **end procedure**

- The objective $f(x)$ can be interpreted as the Euclidean norm $f(\phi) = \phi^\top\phi$ but pulled back into the x-space. More precisely: Consider a mapping $\phi : \mathbb{R}^n \to \mathbb{R}^m$ and a general scalar product $\langle\cdot,\cdot\rangle_\phi$ in the output space. In differential geometry there is the notion of a pull-back of a metric, that is, we define a scalar product $\langle\cdot,\cdot\rangle_x$ in the input space as

$$\langle x, y\rangle_x = \langle d\phi(x), d\phi(y)\rangle_\phi \tag{149}$$

where $d\phi$ is the differential of ϕ (a $\mathbb{R}^m$-valued 1-form). Assuming ϕ-coordinates such that the metric tensor of $\langle\cdot,\cdot\rangle_\phi$ is Euclidean, we have

$$\langle x, x\rangle_x = \langle d\phi(x), d\phi(x)\rangle_\phi = \frac{\partial}{\partial x}\phi(x)^\top \frac{\partial}{\partial x}\phi(x) \tag{150}$$

and therefore, **the approximate Hessian is the pullback of a Euclidean cost feature metric**, and $\langle x, x\rangle_x$ approximates the 2-order polynomial term of $f(x)$, with the non-constant (i.e., Riemannian) pull-back metric $\langle\cdot,\cdot\rangle_x$.

4.1.5 Quasi-Newton & BFGS: approximating the hessian from gradient observations

To apply full Newton methods we need to be able to compute $f(x), \nabla f(x)$, and $\nabla^2 f(x)$ for any x. However, sometimes, computing $\nabla^2 f(x)$ is not possible, e.g., because we cannot derive an analytic expression for $\nabla^2 f(x)$, or it would be too expensive to compute the hessian exactly, or even to store it in memory—especially in very high-dimensional spaces. In such cases it makes sense to approximate $\nabla^2 f(x)$ or $\nabla^2 f(x)^{-1}$ with a low-rank approximation.

Assume we have computed $\nabla f(x_1)$ and $\nabla f(x_2)$ at two different points $x_1, x_2 \in \mathbb{R}^n$. We define

$$y = \nabla f(x_2) - \nabla f(x_1) \ , \quad \delta = x_2 - x_1 \ . \tag{151}$$

From this we may wish to find some approximate Hessian matrix H or H^{-1} that fulfils

$$H\ \delta \stackrel{!}{=} y \qquad \text{or} \qquad \delta \stackrel{!}{=} H^{-1} y \tag{152}$$

The first equation is called *secant equation*. Here are guesses of H and H^{-1}:

$$H = \frac{yy^\top}{y^\top\delta} \qquad \text{or} \qquad H^{-1} = \frac{\delta\delta^\top}{\delta^\top y} \tag{153}$$

Convince yourself that these choices fulfil the respective desired relation above. However, these choices are under-determined. There exist many alternative H or H^{-1} that would be consistent with the observed change in gradient. However, given our understanding of the structure of matrices it is clear that these choices are the lowest rank solutions, namely rank 1.

Broyden-Fletcher-Goldfarb-Shanno (BFGS): An optimization algorithm computes $\nabla f(x_k)$ at a series of points $x_{0:K}$. We incrementally update our guess of H^{-1} with an update equation

$$H^{-1} \leftarrow \left(\mathbf{I} - \frac{y\delta^\top}{\delta^\top y}\right)^\top H^{-1} \left(\mathbf{I} - \frac{y\delta^\top}{\delta^\top y}\right) + \frac{\delta\delta^\top}{\delta^\top y} \;, \tag{154}$$

which is equivalent to (using the Sherman-Morrison formula)

$$H \leftarrow H - \frac{H\delta\delta^\top H^\top}{\delta^T H \delta} + \frac{yy^\top}{y^\top \delta} \;. \tag{155}$$

Note:

- If H^{-1} is initially zero, this update will assign $H^{-1} \leftarrow \frac{\delta\delta^\top}{\delta^\top y}$, which is the minimal rank 1 update we discussed above.
- If H^{-1} is previously non-zero, the red part "deletes certain dimensions" from H^{-1}. More precisely, note that $\left(\mathbf{I} - \frac{y\delta^\top}{\delta^\top y}\right) y = 0$, that is, this rank $n-1$ construction deletes $\text{span}\{y\}$ from its input space. Therefore, the red part gives zero when multiplied with y; and it is guaranteed that the resulting H^{-1} fulfils $H^{-1}y = \delta$.

The **BFGS** algorithms uses this H^{-1} instead of a precise $\nabla^2 f(x)^{-1}$ to compute the steps in a Newton method. All we said about line search and Levenberg-Marquardt damping is unchanged. [6]

In very high-dimensional spaces we do not want to store H^{-1} densely. Instead we use a compressed storage for low-rank matrices, e.g., storing vectors $\{v_i\}$ such that $H^{-1} = \sum_i v_i v_i^\top$. **Limited memory BFGS (L-BFGS)** makes this more memory efficient: it limits the rank of the H^{-1} and thereby the used memory. I do not know the details myself, but I assume that with every update it might aim to delete the lowest eigenvalue to keep the rank constant.

4.1.6 Conjugate Gradient

The Conjugate Gradient Method is a method for solving large linear eqn. systems $Ax + b = 0$. We only mention its extension for optimizing nonlinear functions $f(x)$.

As above, assume that we evaluted $\nabla f(x_1)$ and $\nabla f(x_2)$ at two different points $x_1, x_2 \in \mathbb{R}^n$. But now we make one more assumption: The point x_2 is the minimum of a line search from x_1 along the direction δ_1. This latter assumption is quite optimistic: it

[6] Taken from Peter Blomgren's lecture slides: terminus.sdsu.edu/SDSU/Math693a_f2013/Lectures/18/lecture.pdf This is the original Davidon-Fletcher-Powell (DFP) method suggested by W.C. Davidon in 1959. The original paper describing this revolutionary idea – the first quasi-Newton method – was not accepted for publication. It later appeared in 1991 in the first issue the the SIAM Journal on Optimization.

assumes we did perfect line search. But it gives great information: The iso-lines of $f(x)$ at x_2 are tangential to δ_1.

In this setting, convince yourself of the following: Ideally each search direction should be orthogonal to the previous one—but not orthogonal in the conventional Euclidean sense, but orthogonal w.r.t. the Hessian H. Two vectors d and d' are called **conjugate** w.r.t. a metric H iff $d'^\top H d = 0$. Therefore, subsequent search directions should be conjugate to each other.

Conjugate gradient descent does the following:

Algorithm 5 Conjugate gradient descent

Input: initial $x \in \mathbb{R}^n$, functions $f(x), \nabla f(x)$, tolerance θ
Output: x

1: initialize descent direction $d = g = -\nabla f(x)$
2: **repeat**
3: $\quad \alpha \leftarrow \text{argmin}_\alpha f(x + \alpha d)$ *// line search*
4: $\quad x \leftarrow x + \alpha d$
5: $\quad g' \leftarrow g, \;\; g = -\nabla f(x)$ *// store and compute grad*
6: $\quad \beta \leftarrow \max\left\{ \frac{g^\top (g - g')}{g'^\top g'}, 0 \right\}$
7: $\quad d \leftarrow g + \beta d$ *// conjugate descent direction*
8: **until** $|\Delta x| < \theta$

- The equation for β is by Polak-Ribière: On a quadratic function $f(x) = x^\top H x$ this leads to **conjugate** search directions, $d'^\top H d = 0$.
- Intuitively, $\beta > 0$ implies that the new descent direction always adds a bit of the old direction. This essentially provides 2nd order information.
- For arbitrary quadratic functions CG converges in n iterations. But this only works with **perfect line search**.

4.1.7 Rprop*

Read through Algorithm 6. Notes on this:

- Stepsize adaptation is done in each coordinate *separately*!
- The algorithm not only ignores $|\nabla f|$ but also its exact direction! Only the gradient signs in each coordinate are relevant. Therefore, the step directions may differ up to $< 90^\circ$ from $-\nabla f$.
- It often works surprisingly efficient and robust.

Algorithm 6 Rprop

Input: initial $x \in \mathbb{R}^n$, function $f(x), \nabla f(x)$, initial stepsize α, tolerance θ
Output: x

initialize $x = x_0$, all $\alpha_i = \alpha$, all $g_i = 0$
repeat
  $g \leftarrow \nabla f(x)$
  $x' \leftarrow x$
  for $i = 1 : n$ **do**
    if $g_i g'_i > 0$ **then** *// same direction as last time*
      $\alpha_i \leftarrow 1.2\alpha_i$
      $x_i \leftarrow x_i - \alpha_i \ \text{sign}(g_i)$
      $g'_i \leftarrow g_i$
    else if $g_i g'_i < 0$ **then** *// change of direction*
      $\alpha_i \leftarrow 0.5\alpha_i$
      $x_i \leftarrow x_i - \alpha_i \ \text{sign}(g_i)$
      $g'_i \leftarrow 0$ *// force last case next time*
    else
      $x_i \leftarrow x_i - \alpha_i \ \text{sign}(g_i)$
      $g'_i \leftarrow g_i$
    end if
    optionally: cap $\alpha_i \in [\alpha_{\text{min}} \ x_i, \alpha_{\text{max}} \ x_i]$
  end for
until $|x' - x| < \theta$ for 10 iterations in sequence

- If you like, have a look at:

  Christian Igel, Marc Toussaint, W. Weishui (2005): Rprop using the natural gradient compared to Levenberg-Marquardt optimization. In Trends and Applications in Constructive Approximation. International Series of Numerical Mathematics, volume 151, 259-272.

4.2 The general optimization problem – a mathematical program

Definition 4.1. Let $x \in \mathbb{R}^n$, $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^m$, $h : \mathbb{R}^n \to \mathbb{R}^l$. An optimization problem, or *mathematical program*, is

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & g(x) \le 0 \ , \quad h(x) = 0 \end{aligned}$$

We typically at least assume f, g, h to be differentiable or smooth.

Get an intuition about this problem formulation by considering the following examples. Always discuss where is the optimum, and at the optimum, how the objective f *pulls* at the point, while the constraints g or h *push* against it.

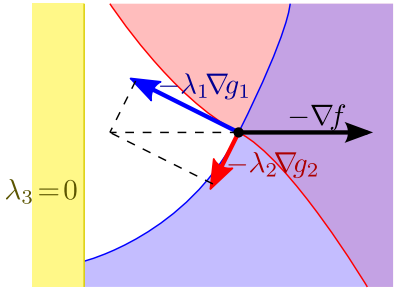


Figure 4: 2D example: $f(x,y) = -x$, pulling constantly to the right; three inequality constraints, two active, one inactive. The "pull/push" vectors fulfil the stationarity condition $\nabla f + \lambda_1 \nabla g_1 + \lambda_2 \nabla g_2 = 0$.

For the following examples, draw the situation and guess, without much maths, where the optimum is:

- A 1D example: $x \in \mathbb{R}$, $h(x) = \sin(x)$, $g(x) = x^2/4 - 1$, f some non-linear function.
- 2D example: $f(x,y) = x$ (intuition: the objective is constantly pulling to the left), $h(x,y) = 0$ is some non-linear path in the plane $\rightarrow$ the optimum is at the left-most tangent-point of h. Tangent-point means that the tangent of h is vertical. h pushes to the right (always orthogonal to the zero-line of h).
- 2D example: $f(x,y) = x$, $g(x,y) = y^2 - x - 1$. The zero-line of g is a parabola towards the right. The objective f pulls into this parabola; the optimum is in the 'bottom' of the parabola, at $(-1, 0)$.
- 2D example: $f(x,y) = x$, $g(x,y) = x^2 + y^2 - 1$. The zero-line of g is a circle. The objective f pulls to the left; the optimum is at the left tangent-point of the circle, at $(-1, 0)$.
- Figure 4

4.3 The KKT conditions

Theorem 4.2 (Karush-Kuhn-Tucker conditions)**.** Given a mathematical program,

$$x \text{ optimal} \quad \Rightarrow \quad \exists \lambda \in \mathbb{R}^m, \kappa \in \mathbb{R}^l \text{ s.t.}$$

$$\nabla f(x) + \sum_{i=1}^{m} \lambda_i \nabla g_i(x) + \sum_{j=1}^{l} \kappa_j \nabla h_j(x) = 0 \qquad \text{(stationarity)}$$

$$\forall_j : \; h_j(x) = 0 \;, \quad \forall_i : \; g_i(x) \leq 0 \qquad \text{(primal feasibility)}$$

$$\forall_i:\ \lambda_i \geq 0 \qquad \text{(dual feasibility)}$$
$$\forall_i:\ \lambda_i g_i(x) = 0 \qquad \text{(complementarity)}$$

Note that these are, in general, only necessary conditions. Only in special cases, e.g. convex, these are also sufficient.

These conditions should be intuitive in the previous examples:

- *The first condition describes the "force balance" of the objective pulling and the active constraints pushing back.* The existance of dual parameters λ, κ could implicitly be expressed by stating

$$\nabla f(x) \in \text{span}(\{\nabla g_{1..m}, \nabla h_{1..l}\}) \tag{156}$$

  The specific values of λ and κ tell us, how strongly the constraints push against the objective, e.g., $\lambda_i |\nabla g_i|$ is the force excerted by the ith inequality.

- *The fourth condition very elegantly describes the logic of inequality constraints being either active ($\lambda_i > 0, g_i = 0$) or inactive ($\lambda_i = 0, g_i \leq 0$).* Intuitively it says: An inequality can only push at the boundary, where $g_i = 0$, but not inside the feasible region, where $g_i < 0$. The trick of using the equation $\lambda_i g_i = 0$ to express this logic is beautiful, especially when later we discuss a case which relaxes this strict logic to $\lambda_i g_i = -\mu$ for some small μ—which roughly means that inequalities may push a little also inside the feasible region.

- Special case $m = l = 0$ (no constraints). The first condition is just the usual $\nabla f(x) = 0$.

- Discuss the previous examples as special cases; and how the force balance is met.

4.4 Unconstrained problems to tackle a constrained problem

Assume you'd know about basic unconstrained optimization methods (like standard gradient descent or the Newton method) but nothing about constrained optimization methods. How would you solve a constrained problem? Well, I think you'd very quickly have the idea to introduce extra cost terms for the violation of constraints—a million people have had this idea and successfully applied it in practice.

In the following we define a new cost function $F(x)$, which includes the objective $f(x)$ and some extra terms.

Definition 4.2 (Log barrier, squared penalty, Lagrangian, Augmented Lagrangian)**.**

$$F_{sp}(x; \nu, \mu) = f(x) + \nu \sum_j h_j(x)^2 + \mu \sum_i [g_i(x) > 0]\ g_i(x)^2 \quad \text{(sqr. penalty)}$$

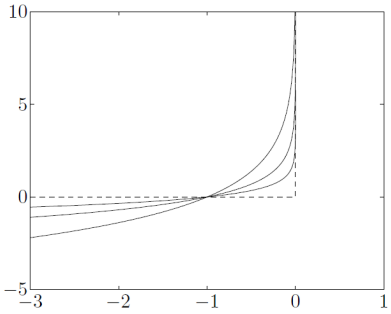


Figure 5: The function $-\mu \log(-g)$ (with g on the "x-axis") for various μ. This is always undefined ("∞") for $g > 0$. For $\mu \to 0$ this becomes the hard step function.

$$F_{lb}(x;\mu) = f(x) - \mu \sum_i \log(-g_i(x)) \qquad \text{(log barrier)}$$

$$L(x,\lambda,\kappa) = f(x) + \sum_j \kappa_j h_j(x) + \sum_i \lambda_i g_i(x) \qquad \text{(Lagrangian)}$$

$$\hat{L}(x) = f(x) + \sum_j \kappa_j h_j(x) + \sum_i \lambda_i g_i(x) + \\ + \nu \sum_j h_j(x)^2 + \mu \sum_i [g_i(x) > 0]\ g_i(x)^2 \qquad \text{(Aug. Lag.)}$$

- The squared penalty method is straight-forward if we have an algorithm to minimize $F(x)$. We initialize $\nu = \mu = 1$, minimize $F(x)$, then increase ν, μ (multiply with a number > 1) and iterate. For $\nu, \mu \to \infty$ we retrieve the optimum.

- The log barrier method (see Fig. 5) does exactly the same, except that we decrease μ towards zero (multiply with a numer < 1 in each iteration). Note that we need a feasible initialization x_0, because otherwise the barriers are ill-defined! The whole algorithm will keep the temporary solutions always *inside* the feasible regions (because the barriers push away from the constraints). That's why it is also called **interior point method**.

- The Lagrangian is a function $L(x,\lambda,\kappa)$ which has the gradient

$$\nabla L(x,\lambda,\kappa) = \nabla f(x) + \sum_i \lambda_i \nabla g_i(x) + \sum_j \kappa_j \nabla h_j(x)\ . \qquad (157)$$

That is, $\nabla L(x,\lambda,\kappa) = 0$ *is our first KKT condition!* In that sense, the additional terms in the Lagrangian *generate the push forces of the constraints*. If we knew the correct λ's and κ's beforehand, then we could find the optimal x by the unconstrained problem $\min_x L(x,\lambda,\kappa)$ (if this has a unique solution).

- The Augmented Lagrangian $\hat{L}$ is a function that includes both, squared penalties, and Lagrangian terms that push proportional to λ, κ. The Augmented Lagrangian method is an iterative algorithm that, while running, figures out how strongly we need to push to ensure that the final solution is *exactly* on the constraints, where all squared penalties will anyway be zero. It does not need to increase ν, μ and still converges to the correct solution.

4.4.1 Augmented Lagrangian*

This is not a main-stream algorithm, but I like it. See Toussaint (2014).

In the Augmented Lagrangian $\hat{L}$, the solver has two types of knobs to tune: the strenghts of the penalties ν, μ and the strengths of the Lagrangian forces λ, κ. The trick is conceptually easy:

- Initially we set $\lambda, \kappa = 0$ and $\nu, \mu = 1$ (or some other constant). In the first iteration, the unconstrained solver will find $x' = \min_x \hat{L}(x)$; the objective f will typically pull into the penalizations.
- For the second iteration we then choose parameters λ, κ that try to avoid that we will be pulled into penalizations the next time. Let's update

$$\kappa_j \leftarrow \kappa_j + 2\mu h_j(x') \ , \quad \lambda_i \leftarrow \max(\lambda_i + 2\mu g_i(x'), 0). \tag{158}$$

  Note that $2\mu h_j(x')$ is the force (gradient) of the equality penalty at x'; and $\max(\lambda_i + 2\mu g_i(x'), 0)$ is the force of the inequality constraint at x'. What this update does is: it analyzes the forces excerted by the penalties, and translates them to forces excerted by the Lagrange terms in the next iteration. It tries to trade the penalizations for the Lagrange terms.

  More rigorously, observe that, if f, g, h are linear and the same constraints are active in two consecutive iterations, then this update will guarantee that all penalty terms are zero in the second iteration, and therefore the solution fulfils the first KKT condition (Toussaint, 2014). See also the respective exercise.

4.5 The Lagrangian

4.5.1 How the Lagrangian relates to the KKT conditions

The Lagrangian $L(x, \kappa, \lambda) = f + \kappa^\top h + \lambda^\top g$ has a number of properties that relates it to the KKT conditions:

(i) Requiring a zero-x-gradient, $\nabla_x L = 0$, implies the *1st KKT condition*.

(ii) Requiring a zero-κ-gradient, $0 = \nabla_\kappa L = h$, implies primal feasibility (the *2nd KKT condition*) w.r.t. the equality constraints.

(iii) Requiring that L is maximized w.r.t. $\lambda \geq 0$ is related to the remaining 2nd and 4th KKT conditions:

$$\max_{\lambda \geq 0} L(x, \lambda) = \begin{cases} f(x) & \text{if } g(x) \leq 0 \\ \infty & \text{otherwise} \end{cases} \tag{159}$$

$$\lambda = \operatorname*{argmax}_{\lambda \geq 0} L(x, \lambda) \quad \Rightarrow \quad \begin{cases} \lambda_i = 0 & \text{if } g_i(x) < 0 \\ 0 = \nabla_{\lambda_i} L(x, \lambda) = g_i(x) & \text{otherwise} \end{cases} \tag{160}$$

This implies either $(\lambda_i = 0 \wedge g_i(x) < 0)$ or $g_i(x) = 0$, which is equivalent to the *complementarity* and *primal feasibility* for inequalities.

These three facts show how tightly the Lagrangian is related to the KKT conditions. To simplify the discussion let us assume only inequality constraints from now on. Fact (i) tells us that if we $\min_x L(x, \lambda)$, we reproduce the 1st KKT condition. Fact (iii) tells us that if we $\max_{\lambda \geq 0} L(x, \lambda)$, we reproduce the remaining KKT conditions. Therefore, the optimal primal-dual solution (x^*, λ^*) can be characterized as a **saddle point of the Lagrangian**. Finding the saddle point can be written in two ways:

Definition 4.3 (primal and dual problem)**.**

$$\min_x \max_{\lambda \geq 0} L(x, \lambda) \qquad \textbf{(primal problem)}$$

$$\max_{\lambda \geq 0} \underbrace{\min_x L(x, \lambda)}_{l(\lambda)} \qquad \textbf{(dual problem)}$$

$$l(\lambda) \quad \textbf{(dual function)}$$

Convince yourself, using 159, that the first expression is indeed the original primal problem $\Big[\min_x f(x) \;\; \text{s.t.} \;\; g(x) \leq 0\Big]$.

What can we learn from this? The KKT conditions state that, at an optimum, *there exist some* λ, κ. This existance statement is not very helpful to actually find them. In contrast, the Lagrangian tells us directly how the dual parameters can be found: by maximizing w.r.t. them. This can be exploited in several ways:

4.5.2 Solving mathematical programs analytically, on paper.

Consider the problem

$$\min_{x \in \mathbb{R}^2} x^2 \;\; \text{s.t.} \;\; x_1 + x_2 = 1 \; . \tag{161}$$

We can find the solution analytically via the Lagrangian:

$$L(x, \kappa) = x^2 + \kappa(x_1 + x_2 - 1) \tag{162}$$

$$0 = \nabla_x L(x,\kappa) = 2x + \kappa \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \Rightarrow \quad x_1 = x_2 = -\kappa/2 \tag{163}$$

$$0 = \nabla_\kappa L(x,\kappa) = x_1 + x_2 - 1 = -\kappa/2 - \kappa/2 - 1 \quad \Rightarrow \quad \kappa = -1 \tag{164}$$

$$\Rightarrow x_1 = x_2 = 1/2 \tag{165}$$

Here we first formulated the Lagrangian. In this context, κ is often called **Lagrange multiplier**, but I prefer the term *dual variable*. Then we find a saddle point of L by requiring $0 = \nabla_x L(x,\kappa)$, $0 = \nabla_\kappa L(x,\kappa)$. If we want to solve a problem with an inequality constrained, we do the same calculus for both cases: 1) the constraint is active (handled like an equality constrained), and 2) the constrained is inactive. Then we check if the inactive case solution is feasible, or the active case is dual-feasible ($\lambda \geq 0$). Note that if we have m inequality constraints we have to analytically evaluate every combination of constraints being active/inactive—which are 2^m cases. This already hints at the fact that a real difficulty in solving mathematical programs is to find out which inequality constraints are active or inactive. In fact, if we knew this a priori, everything would reduce to an equality constrained problem, which is much easier to solve.

4.5.3 Solving the dual problem, instead of the primal.

In some cases the dual function $l(\lambda) = \min_x L(x,\lambda)$ can analytically be derived. In this case it makes very much sense to try solving the dual problem instead of the primal. First, the dual problem $\max_{\lambda\geq 0} l(\lambda)$ is guaranteed to be convex even if the primal is non-convex. (The dual function $l(\lambda)$ is concave, and the constraint $\lambda \geq 0$ convex.) But note that $l(\lambda)$ is itself defined as the result of a generally non-convex optimization problem $\min_x L(x,\lambda)$. Second, the inequality constraints of the dual problem are very simple: just $\lambda \geq 0$. Such inequality constraints are called **bound constraints** and can be handled with specialized methods.

However, in general $\min_x \max_y f(x,y) \neq \max_y \min_x f(x,y)$. For example, in discrete domain $x, y \in \{1,2\}$, let $f(1,1) = 1, f(1,2) = 3, f(2,1) = 4, f(2,2) = 2$, and $\min_x f(x,y) = (1,2)$ and $\max_y f(x,y) = (3,4)$. Therefore, the dual problem is in general not equivalent to the primal.

The dual function is, for $\lambda \geq 0$, a lower bound

$$l(\lambda) = \min_x L(x,\lambda) \;\leq\; \Big[\min_x f(x) \;\; \text{s.t.} \;\; g(x) \leq 0\Big] \,. \tag{166}$$

And consequently

$$\text{(dual)} \qquad \max_{\lambda\geq 0} \min_x L(x,\lambda) \;\leq\; \min_x \max_{\lambda\geq 0} L(x,\lambda) \qquad \text{(primal)} \tag{167}$$

We say **strong duality** holds iff

$$\max_{\lambda\geq 0} \min_x L(x,\lambda) = \min_x \max_{\lambda\geq 0} L(x,\lambda) \tag{168}$$

If the primal is convex, and there exist an interior point

$$\exists_x : \ \forall_i : \ g_i(x) < 0 \tag{169}$$

(which is called **Slater condition**), then we have **strong duality**.

4.5.4 Finding the "saddle point" directly with a primal-dual Newton method.

In basic unconstrained optimization an efficient way to find an optimum (minimum or maximum) is to find a point where the gradient is zero with a Newton method. At saddle points all gradients are also zero. So, to find a saddle point of the Lagrangian we can equally use a Newton method that seeks for roots of the gradient. Note that such a Newton method optimizes in the joint **primal-dual space** of (x, λ, κ).

In the case of inequalities, the zero-gradients view is over-simplified: While facts (i) and (ii) characterize a saddle point in terms of zero gradients, fact (iii) makes this more precise to handle the inequality case. For this reason it is actually easier to describe the primal-dual Newton method directly in terms of the KKT conditions: We seek a point (x, λ, κ), with $\lambda \geq 0$, that solves the equation system

$$\nabla_x f(x) + \lambda^\top \partial_x g + \kappa^\top \partial_x h = 0 \tag{170}$$

$$h(x) = 0 \tag{171}$$

$$\text{diag}(\lambda) g(x) + \mu \mathbf{1}_m = 0 \tag{172}$$

Note that the first equation is the 1st KKT, the 2nd is the 2nd KKT w.r.t. equalities, and the third is the *approximate* 4th KKT with log barrier parameter μ (see below). These three equations reflect the saddle point properties (facts (i), (ii), and (iii) above). We define

$$r(x, \lambda, \kappa) = \begin{pmatrix} \nabla f(x) + \lambda^\top \partial g(x) + \kappa^\top \partial h(x) \\ h(x) \\ \text{diag}(\lambda)\ g(x) + \mu \mathbf{1}_m \end{pmatrix} \tag{173}$$

and use the Newton method

$$\begin{pmatrix} x \\ \lambda \\ \kappa \end{pmatrix} \leftarrow \begin{pmatrix} x \\ \lambda \\ \kappa \end{pmatrix} - \alpha \ \partial r(x, \lambda, \kappa)^{-1} \ r(x, \lambda) \tag{174}$$

to find the root $r(x, \lambda, \kappa) = 0$ (α is the stepsize). We have

$$\partial r(x, \lambda, \kappa) = \begin{pmatrix} \nabla^2 f(x) + \sum_i \lambda_i \nabla^2 g_i(x) + \sum_j \kappa_j \nabla^2 h_j(x) & \partial g(x)^\top & \partial h(x)^\top \\ \partial h(x) & 0 & 0 \\ \text{diag}(\lambda)\ \partial g(x) & \text{diag}(g(x)) & 0 \end{pmatrix} \tag{175}$$

where $\partial r(x, \lambda, \kappa) \in \mathbb{R}^{(n+m+l)\times(n+m+l)}$. Note that this method uses the hessians $\nabla^2 f, \nabla^2 g$ and $\nabla^2 h$.

The above formulation allows for a duality gap μ. One could choose $\mu = 0$, but often that is not robust. The beauty is that we can adapt μ on the fly, before each Newton step, so that we do not need a separate outer loop to adapt μ.

Before computing a Newton step, we compute the current duality measure $\tilde{\mu} = -\frac{1}{m} \sum_{i=1}^{m} \lambda_i g_i(x)$. Then we set $\mu = \frac{1}{2}\tilde{\mu}$ to half of this. In this way, the Newton step will compute a direction that aims to half the current duality gap. In practise, this leads to good convercence in a single-loop Newton method. (See also Boyd sec 11.7.3.)

The dual feasibility $\lambda_i \geq 0$ needs to be handled explicitly by the root finder – the line search can simply clip steps to stay within the bound constraints.

Typically, the method is called "*interior* primal-dual Newton", in which case also the primal feasibility $g_i \leq 0$ has to be ensured. But I found there are tweaks to make the method also handle infeasible x, including infeasible initializations.

4.5.5 Log Barriers and the Lagrangian

Finally, let's revisit the log barrier method. In principle it is very simple: For a given μ, we use an unconstrained solver to find the minimum $x^*(\mu)$ of

$$F(x;\mu) = f(x) - \mu \sum_i \log(-g_i(x)) \ . \tag{176}$$

(This process is also called "centering".) We then gradually decrease μ to zero, always calling the inner loop to recenter. The generated path of $x^*(\mu)$ is called central path.

The method is simple and has very insightful relations to the KKT conditions and the dual problem. For given μ, the optimality condition is

$$\nabla F(x;\mu) = 0 \quad \Rightarrow \quad \nabla f(x) - \sum_i \frac{\mu}{g_i(x)} \nabla g_i(x) = 0 \tag{177}$$

$$\Leftrightarrow \quad \nabla f(x) + \sum_i \lambda_i \nabla g_i(x) = 0 \ , \quad \lambda_i g_i(x) = -\mu \tag{178}$$

where we defined(!) $\lambda_i = -\mu/g_i(x)$, which guarantees $\lambda_i \geq 0$ as long as we are in the interior ($g_i \leq 0$).

So, $\nabla F(x;\mu) = 0$ is equivalent to the **modified (=approximate) KKT conditions**, where the complemenetarity is relaxed: inequalities may push also inside the feasible region. For $\mu \to 0$ we converge to the exact KKT conditions with strict complementarity.

So μ has the interpretation of a relaxation of complementarity. We can derive another interpretation of μ in terms of suboptimality or the duality gap:

Let $x^*(\mu) = \min_x F(x;\mu)$ be the central path. At each x^* we may define, as above, $\lambda_i = -\mu/g_i(x^*)$. We note that $\lambda \geq 0$ (dual feasible), as well that $x^*(\mu)$ minimizes the

Lagrangian $L(x, \lambda)$ w.r.t. x! This is because,

$$0 = \nabla F(x, \mu) = \nabla f(x) + \sum_{i=1}^{m} \lambda_i \nabla g_i(x) = \nabla L(x, \lambda) \ . \tag{179}$$

Therefore, x^* is actually the solution to $\min_x L(x, \lambda)$, which defines the dual function. We have

$$l(\lambda) = \min_x L(x, \lambda) = f(x^*) + \sum_{i=1}^{m} \lambda_i g_i(x^*) = f(x^*) - m\mu \ . \tag{180}$$

(m is simply the count of inequalities.) That is, $m\mu$ is the duality gap between the (suboptimal) $f(x^*)$ and $l(\lambda)$. Further, given that the dual function is a lower bound, $l(\lambda) \leq p^*$, where $p^* = \min_x f(x)$ s.t. $g(x) \leq 0$ is the optimal primal value, we have

$$f(x^*) - p^* \leq m\mu \ . \tag{181}$$

This gives the interpretation of μ as an upper bound on the suboptimality of $f(x^*)$.

4.6 Convex Problems

We do not put much emphasis on discussing convex problems in this lecture. The algorithms we discussed so far equally apply on general non-linear programs as well as on convex problems—of course, only on convex problems we have convergence guarantees, as we can see from the convergence rate analysis of Wolfe steps based on the assumption of positive upper and lower bounds on the Hessian's eigenvalues.

Nevertheless, we at least define standard LPs, QPs, etc. Perhaps the most interesting part is the discussion of the Simplex algorithm—not because the algorithms is nice or particularly efficient, but rather because one gains a lot of insights in what actually makes (inequality) constrained problems hard.

4.6.1 Convex sets, functions, problems

Definition 4.4 (Convex set, convex function)**.** A set $X \subseteq V$ (a subset of some vector space V) is **convex** iff

$$\forall x, y \in X, a \in [0,1] : \ ax + (1-a)y \in X \tag{182}$$

A function is defined

$$\textbf{convex} \ \Leftrightarrow \ \forall x, y \in \mathbb{R}^n, a \in [0,1] : \ f(ax + (1-a)y) \leq a \ f(x) + (1-a) \ f(y) \tag{183}$$

$$\textbf{quasiconvex} \ \Leftrightarrow \ \forall x, y \in \mathbb{R}^n, a \in [0,1] : \ f(ax + (1-a)y) \leq \max\{f(x), f(y)\} \tag{184}$$

Note: quasiconvex $\Leftrightarrow$ for any $\alpha \in \mathbb{R}$ the sublevel set $\{x | f(x) \leq \alpha\}$ is convex. Further, I call a function **unimodal** if it has only one local minimum, which is the global minimum.

Definition 4.5 (Convex program)**.**

Variant 1: A mathematical program $\min_x \ f(x)$ s.t. $g(x) \leq 0,\ h(x) = 0$ is convex if f is convex and the feasible set is convex.

Variant 2: A mathematical program $\min_x \ f(x)$ s.t. $g(x) \leq 0,\ h(x) = 0$ is convex if f and every g_i are convex and h is linear.

Variant 2 is the stronger and usual definition. Concerning variant 1, if the feasible set is convex the zero-level sets of all g's need to be convex and the zero-level sets of h's needs to be linear. Above these zero levels the g's and h's could in principle be abribtrarily non-linear, but these non-linearities are irrelevant for the mathematical program itself. We could replace such g's and h's by convex and linear functions and get the same problem.

4.6.2 Linear and quadratic programs

Definition 4.6 (Linear program (LP), Quadratic program (QP))**.** Special case mathematical programs are

$$\textbf{Linear Program (LP):} \quad \min_x \ c^\top x \ \text{ s.t. } \ Gx \leq h,\ Ax = b$$

$$\textbf{LP in standard form:} \quad \min_x \ c^\top x \ \text{ s.t. } \ x \geq 0,\ Ax = b$$

$$\textbf{Quadratic Program (QP):} \quad \min_x \ \frac{1}{2} x^\top Q x + c^\top x \ \text{ s.t. } \ Gx \leq h,\ Ax = b \ , Q\text{pos-def}$$

Rarely, also a **Quadratically Constrained QP (QCQP)** is considered.

An important example for LP are **relaxations** of integer linear programs,

$$\min_x c^\top x \ \text{ s.t. } \ Ax = b,\ x_i \in \{0, 1\} \ , \tag{185}$$

which includes Travelling Salesman, MaxSAT or MAP inference problems. Relaxing such a problem means to instead solve the continuous LP

$$\min_x c^\top x \ \text{ s.t. } \ Ax = b,\ x_i \in [0, 1] \ . \tag{186}$$

If one is lucky and the continuous LP problem converges to a fully integer solution, where all $x_i \in \{0, 1\}$, this is also the solution to the integer problem. Typically, the solution of the continuous LP will be partially integer (some values converge to the extreme $x_i \in \{0, 1\}$, while others are inbetween $x_i \in (0, 1)$). This continuous valued solution gives a lower bound on the integer problem, and provides very efficient heuristics for backtracking or branch-and-bound search for a fully integer solution.

The standard example for a QP are Support Vector Machines. The primal problem is

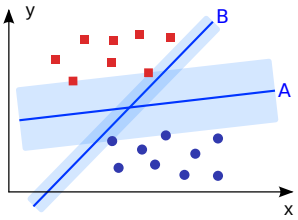
$$\min_{\beta, \xi} \ \|\beta\|^2 + C \sum_{i=1}^{n} \xi_i \ \text{ s.t. } \ y_i (x_i^\top \beta) \geq 1 - \xi_i \ , \quad \xi_i \geq 0 \tag{187}$$

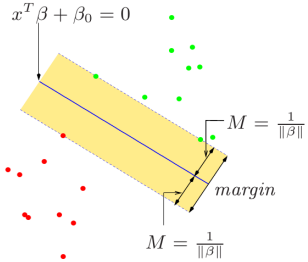
the dual

$$l(\alpha,\mu) = \min_{\beta,\xi} L(\beta,\xi,\alpha,\mu) = -\tfrac{1}{4}\sum_{i=1}^{n}\sum_{i'=1}^{n}\alpha_i\alpha_{i'}y_iy_{i'}\hat{x}_i^\top\hat{x}_{i'} + \sum_{i=1}^{n}\alpha_i \tag{188}$$

$$\max_{\alpha,\mu} l(\alpha,\mu) \quad \text{s.t.} \quad 0 \le \alpha_i \le C \tag{189}$$

(See ML lecture 5:13 for a derivation.)





4.6.3 The Simplex Algorithm

Consider an LP. We make the following observations:

- First, in LPs the equality constraints could be resolved simply by introducing new coordinates along the zero-hyperplane of h. Therefore, for the conceptual discussion we neglect equality constraints.
- The objective constantly pulls in the direction $-c = -\nabla f(x)$.
- If the solution is bounded there need to be some inequality constraints that keep the solution from travelling to ∞ in the $-c$ direction.
- It follows: **The solution will always be located at a vertex**, that is, an intersection point of several zero-hyperplanes of inequality constraints.
- In fact, we should think of the feasible region as a **polytope** that is defined by all the zero-hyperplanes of the inequalities. The inside the polytope is the feasible region. The polytope has edges (intersections of two contraint planes), faces, etc. A solution will always be located at a vertex of the polytope; more precisely, there could be a whole set of optimal points (on a face orthogonal to c), but at least one vertex is also optimal.
- An idea for finding the solution is to **walk on the edges of the polytope** until an optimal vertex is found. This is the simplex algorithm of Georg Dantzig, 1947. In practise this procedure is done by "pivoting on the simplex tableaux"—but we fully skip such details here.

- The simplex algorithm is often efficient, but in worst case it is exponential in both, n and m! This is hard to make intuitive, because the effects of high dimensions are not intuitive. But roughly, consider that in high dimensions there is a combinatorial number of ways of how constraints may intersect and form edges and vertices.

Here is a view that much more relates to our discussion of the log barrier method: Sitting on an edge/face/vertex is equivalent to temporarily deciding which constraints are active. If we knew which constraints are eventually active, the problem would be solved: all inequalities become equalities or void. (And linear equalities can directly be solved for.) So, jumping along vertices of the polytope is equivalent to sequentially making decisions on which constraints might be active. Note though that there are 2^m configurations of active/non-active constraints. The simplex algorithm therefore walks through this combinatorial space.

Interior point methods do exactly the opposite: Recall that the **4th KKT condition** is $\lambda_i g_i(x) = 0$. The log barrier method (for instance) instead *relaxes* this hard logic of activ/non-active constraints and finds in each iteration a solution to the relaxed 4th KKT condition $\lambda_i g_i(x) = -\mu$, which intuitively means that every constraint may be "somewhat active". In fact, every constraint contributes somewhat to the stationarity condition via the log barrier's gradients. Thereby interior point methods

- post-pone the hard decisions about active/non-active constraints
- approach the optimal vertex from the inside of the polytope; avoiding the polytope surface (and its hard decisions)
- thereby avoids the need to search through a combinatorial space of constraint activities and instead continuously converges to a decision
- has polynomial worst-case guaranteed

Historically, penalty and barrier methods methods were standard before the Simplex Algorithm. When SA was discovered in the 50ies, it was quickly considered great. But then, later in the 70-80ies, a lot more theory was developed for interior point methods, which now again have become somewhat more popular than the simplex algorithm.

4.6.4 Sequential Quadratic Programming

Just for reference, SQP is another standard approach to solving non-linear mathematical programs. In each iteration we compute all coefficients of the 2nd order Taylor $f(x+\delta) \approx f(x) + \nabla f(x)^\top \delta + \frac{1}{2}\delta^\top H \delta$ and 1st-order Taylor $g(x+\delta) \approx g(x) + \nabla g(x)^\top \delta$ and then solve the QP

$$\min_\delta \; f(x) + \nabla f(x)^\top \delta + \frac{1}{2}\delta^\top \nabla^2 f(x) \delta \;\; \text{s.t.} \;\; g(x) + \nabla g(x)^\top \delta \leq 0 \tag{190}$$

The optimal δ^* of this problem should be seen analogous to the optimal Newton step: If f *were* a 2nd-order polynomial and g linear, then δ^* would jump directly to the optimum. However, as this is generally not the case, δ^* only gives us a very good direction for line search. In SQP, we need to backtrack until we found a feasible point *and* f decreases sufficiently.

Solving each QP in the sub-routine requires a constrained solver, which itself might have two nested loops (e.g. using log-barrier or AugLag). In that case, SQP has three nested loops.

4.7 Blackbox & Global Optimization: It's all about learning

Even if f, g, h are smooth, the solver might not have access to analytic equations or efficient numeric methods to evaluate the gradients or hessians of these. Therefore we distinguish (here neglecting the constraint functions g and h):

Definition 4.7.

- *Blackbox optimization:* Only $f(x)$ can be evaluated.
- *1st-order/gradient optimization:* Only $f(x)$ and $\nabla f(x)$ can be evaluated.
- *Quasi-Newton optimization:* Only $f(x)$ and $\nabla f(x)$ can be evaluated, but the solver does tricks to estimate $\nabla^2 f(x)$. (So this is a special case of 1st-order optimization.)
- *Gauss-Newton type optimization:* f is of the special form $f(x) = \phi(x)^\top \phi(x)$ and $\frac{\partial}{\partial x}\phi(x)$ can be evaluated.
- *2nd order optimization:* $f(x)$, $\nabla f(x)$ and $\nabla^2 f(x)$ can be evaluated.

In this lecture I very briefly want to add comments on **global** blackbox optimization. Global means that we now, for the first time, aim to find the global optimum (within some pre-specified bounded range). In essence, to address such a problem we need to explicitly know what we know about f[7], and an obvious way to do this is to use Bayesian learning.

4.7.1 A sequential decision problem formulation

From now on, let's neglect constraints and focus on the mathematical program

$$\min_x f(x) \tag{191}$$

[7] Cf. the KWIK (knows what it knows) framework.

for a blackbox function f. The optimization process can be viewed as a Markov Decision Process that describes the interaction of the solver (agent) with the function (environment):

- At step t, $D_t = \{(x_i, y_i)\}_{i=1}^{t\text{-}1}$ is the data that the solver has collected from previous samples. This D_t is the state of the MDP.
- At step t, the solver may choose a new decision x_t about where to sample next.
- Given state D_t and decision x_t, the next state is $D_{t+1} = D \cup \{(x_t, f(x_t))\}$, which is a deterministic transition *given* the function f.
- A solver policy is a mapping $\pi : D_t \mapsto x_t$ that maps any state (of knowledge) to a new decision.
- We may define an **optimal solver policy** as

$$\pi^* = \underset{\pi}{\text{argmin}} \langle y_T \rangle = \underset{\pi}{\text{argmin}} \int_f P(f)\ P(D_T|\pi, f)\ y_T \tag{192}$$

  where $P(D_T|\pi, f)$ is deterministic, and $P(f)$ is a **prior over functions**.

  This objective function cares only about the *last* value y_T sampled by the solver for a fixed time horizon (budget) T. Alternatively, we may choose objectives $\sum_{t=1}^T y_t$ or $\sum_{t=1}^T \gamma^t y_t$ for some discounting $\gamma \in [0, 1]$

The above defined what is an optimal solver! Something we haven't touched at all before. The transition dynamics of this MDP is deterministic, given f. However, from the perspective of the solver, we do not know f apriori. But we can always compute a **posterior belief** $P(f|D_t) = P(D_t|f)\ P(f)/P(D_t)$. This posterior belief defines a **belief MDP** with stochastic transitions

$$P(D_{t+1}) = \int_{D_t} \int_f \int_{x_t} [D_{t+1} = D \cup \{(x_t, f(x_t))\}]\ \pi(x_t|D_t)\ P(f|D_t)\ P(D_t)\ . \tag{193}$$

The belief MDP's state space is $P(D_t)$ (or equivalently, $P(f|D_t)$, the current belief over f). This belief MDP is something that the solver can, in principle, forward simulate—it has all information about it. One can prove that, if the solver could solve its own belief MDP (find an optimal policy for its belief MDP), then this policy is the optimal solver policy for the original problem given a prior distribution $P(f)$! So, in principle we not only defined what is an optimal solver policy, but can also provide an algorithm to compute it (Dynamic programming in the belief MDP)! However, this is so expensive to compute that heuristics need to be used in practise.

One aspect we should learn from this discussion: The solver's optimal decision is based on its current belief $P(f|D_t)$ over the function. This belief is the Bayesian representation of everything one could possibly have learned about f from the data D_t collected so far. Bayesian Global Optimization methods compute $P(f|D_t)$ in every step and, based on this, use a heuristic to choose the next decision.

4.7.2 Acquisition Functions for Bayesian Global Optimization*

In pratice one typically uses a Gaussian Process representation of $P(f|D_t)$. This means that in every iteration we have an estimate $\hat{f}(x)$ of the function mean and a variance estimate $\hat{\sigma}(x)^2$ that describes our uncertainty about the mean estimate. Based on this we may define the following acquisition functions

Definition 4.8. Probability of Improvement (MPI)

$$\alpha_t(x) = \int_{-\infty}^{y^*} \mathcal{N}(y|\hat{f}(x), \hat{\sigma}(x))\ dy \tag{194}$$

Expected Improvement (EI)

$$\alpha_t(x) = \int_{-\infty}^{y^*} \mathcal{N}(y|\hat{f}(x), \hat{\sigma}(x))\ (y^* - y)\ dy \tag{195}$$

Upper Confidence Bound (UCB)

$$\alpha_t(x) = -\hat{f}(x) + \beta_t \hat{\sigma}(x) \tag{196}$$

Predictive Entropy Search Hernández-Lobato et al. (2014)

$$\begin{aligned}\alpha_t(x) &= H[p(x^*|D_t)] - \mathbb{E}\{p(y|D_t; x)\}\, H[p(x^*|D_t \cup \{(x,y)\})] \\ &= I(x^*, y|D_t) = H[p(y|D_t, x)] - \mathbb{E}\{p(x^*|D_t)\}\, H[p(y|D_t, x, x^*)]\end{aligned} \tag{197}$$

The last one is special; we'll discuss it below.

These acquisition functions are heuristics that define how valuable it is to acquire data from the site x. The solver then makes the decision

$$x_t = \operatorname*{argmax}_x \alpha_t(x)\ . \tag{198}$$

MPI is hardly being used in practise anymore. EI is classical, originating way back in the 50ies or earlier; Jones et al. (1998) gives an overview. UCB received a lot of attention recently due to the underlying bandit theory and bounded regret theorems due to the submodularity. But I think that in practise EI and UCB perform about equally. As UCB is somewhat easier to implement and intuitive.

In all cases, note that the solver policy $x_t = \operatorname{argmax}_x \alpha_t(x)$ requires to internally solve another non-linear optimization problem. However, α_t is an analytic function for which we can compute gradients and hessians which ensures every efficient *local* convergence. But again, $x_t = \operatorname{argmax}_x \alpha_t(x)$ needs to be solved *globally*—otherwise the solver will also not solve the original problem properly and globally. As a consequence, the optimization of the acquisition function needs to be restarted from many many potential start points close to potential local minima; typically from grid(!) points over the full domain range. The number of grid points is exponential in the problem dimension n. Therefore, this inner loop can be very expensive.

And a subjective note: This all sounds great, but be aware that Gaussian Processes with standard squared-exponential kernels do not generalize much in high dimensions: one roughly needs exponentially many data points to fully cover the domain and reduce belief uncertainty globally, almost as if we were sampling from a grid with grid size equal to the kernel width. So, the whole approach is not magic. It just does what is possible given a belief $P(f)$. It would be interesting to have much more structured (and heteroscedastic) beliefs specifically for optimization.

The last acquisition function is called **Predictive Entropy Search**. This formulation is beautiful: We sample at places x where the (expected) observed value y informs us as much as possible about the optimum x^* of the function! Formally, this means to maximize the mutual information between y and x^*, in expectation over $y|x$.

4.7.3 Classical model-based blackbox optimization (non-global)*

A last method very worth mentioning: Classical model-based blackbox optimization simply fits a local polynomial model to the recent data and takes this a basis for search. This is similar to BFGS, but now for the blackbox case where we not even observe gradients. See Algorithm 7.

The local fitting of a polynomial model is again a Machine Learning method. Whether this gives a function approximation for optimization depends on the quality of the data D_t used for this approximation. Classical model-based optimization has interesting heuristics to evaluate the data quality as well as sample new points to improve the data quality. Here is a rough algorithm (following Nodecal et al.'s section on "Derivative-free optimization"):

Some notes:

- Line 4 implement an explicit trust region approach, which hard bound α on the step size.
- Line 5 is like the Wolfe condition. But here, the expected decrease is $[f(\hat{x}) - \hat{f}(\hat{x}+\delta)]$ instead of $-\alpha\delta\nabla f(x)$.
- If there is no sufficient decrease we may blame it on two reasons: bad data or a too large stepsize.
- Line 10 uses the *data determinant* as a measure of quality! This is meant in the sense of linear regression on polynomial features. Note that, with data matrix $X \in \mathbb{R}^{n\times \dim(\beta)}$, $\hat{\beta}^{\text{ls}} = (\boldsymbol{X}^\top \boldsymbol{X})^{-1}\boldsymbol{X}^\top y$ is the optimal regression. The determinant $\det(\boldsymbol{X}^\top \boldsymbol{X})$ or $\det(\boldsymbol{X}) = \det(D)$ is a measure for well the data supports the regression. If the determinant is zero, the regression problem is ill-defined. The larger the determinant, the lower the variance of the regression estimator.
- Line 11 is an explicit exploration approach: We add a data point solely for the purpose of increasing the data determinant (increasing the data spread). Interest-

Algorithm 7 Classical model-based optimization

1: Initialize D with at least $\frac{1}{2}(n+1)(n+2)$ data points
2: **repeat**
3: Compute a regression $\hat{f}(x) = \phi_2(x)^\top \beta$ on D
4: Compute $\delta = \text{argmin}_\delta \hat{f}(\hat{x}+\delta)$ s.t. $|\delta| < \alpha$
5: **if** $f(\hat{x}+\delta) < f(\hat{x}) - \varrho_{\text{ls}}[f(\hat{x}) - \hat{f}(\hat{x}+\delta)]$ **then** *// test sufficient decrease*
6: Increase the stepsize α
7: Accept $\hat{x} \leftarrow \hat{x}+\delta$
8: Add to data, $D \leftarrow D \cup \{(\hat{x}, f(\hat{x}))\}$
9: **else** *// no sufficient decrease*
10: **if** $\det(D)$ is too small **then** *// blame the data quality*
11: Compute $x^+ = \text{argmax}_{x'} \det(D \cup \{x'\})$ s.t. $|x - x'| < \alpha$
12: Add to data, $D \leftarrow D \cup \{(x^+, f(x^+))\}$
13: **else** *// blame the stepsize*
14: Decrease the stepsize α
15: **end if**
16: **end if**
17: Perhaps prune the data, e.g., remove $\text{argmax}_{x \in \Delta} \det(D \setminus \{x\})$
18: **until** x converges

ing. Nocedal describes in more detail a geometry-improving procedure to update D.

4.7.4 Evolutionary Algorithms*

There are interesting and theoretically well-grounded evolutionary algorithms for optimization, such as Estimation-of-Distribution Algorithms (EDAs). But generally, don't use them as first choice.

4.8 Examples and Exercises

4.8.1 Convergence proof

a) Given a function $f : \mathbb{R}^n \to \mathbb{R}$ with $f_{\text{min}} = \min_x f(x)$. Assume that its Hessian—that is, the eigenvalues of $\nabla^2 f$—are lower bounded by $m > 0$ and upper bounded by $M > m$, with $m, M \in \mathbb{R}$. Prove that for any $x \in \mathbb{R}^n$ it holds

$$f(x) - \frac{1}{2m}|\nabla f(x)|^2 \le f_{\text{min}} \le f(x) - \frac{1}{2M}|\nabla f(x)|^2 \ .$$

Tip: Start with bounding $f(x)$ between the functions with maximal and minimal curvature. Then consider the minima of these bounds. Note, it also follows:

$$|\nabla f(x)|^2 \ge 2m(f(x) - f_{\text{min}}) \ .$$

b) Consider backtracking line search with Wolfe parameter $\varrho_{\text{ls}} \leq \frac{1}{2}$, and step decrease factor ϱ_α^-. First prove that line search terminates the latest when $\frac{\varrho_\alpha^-}{M} \leq \alpha \leq \frac{1}{M}$, and then it found a new point y for which

$$f(y) \leq f(x) - \frac{\varrho_{\text{ls}}\varrho_\alpha^-}{M}|\nabla f(x)|^2 \ .$$

From this, using the result from a), prove the convergence equation

$$f(y) - f_{\text{min}} \leq \left[1 - \frac{2m\varrho_{\text{ls}}\varrho_\alpha^-}{M}\right] (f(x) - f_{\text{min}}) \ .$$

4.8.2 Backtracking Line Search

Consider the functions

$$f_{\text{sq}}(x) = x^\top C x \ , \tag{199}$$

$$f_{\text{hole}}(x) = 1 - \exp(-x^\top C x) \ . \tag{200}$$

with diagonal matrix C and entries $C(i,i) = c^{\frac{i-1}{n-1}}$, where n is the dimensionality of x. We choose a conditioning[8] $c = 10$. To plot the function for $n = 2$, you can use gnuplot calling

```
set isosamples 50,50
set contour
f(x,y) = x*x+10*y*y
#f(x,y) = 1 - exp(-x*x-10*y*y)
splot [-1:1][-1:1] f(x,y)
```

a) Implement gradient descent with backtracking, as described on page 42 (Algorithm 2 Plain gradient descent). Test the algorithm on $f_{\text{sq}}(x)$ and $f_{\text{hole}}(x)$ with start point $x_0 = (1,1)$. To judge the performance, create the following plots:

- The function value over the number of function evaluations.
- For $n = 2$, the function surface including algorithm's search trajectory. If using gnuplot, store every evaluated point x and function value $f(x)$ in a line (with $n+1$ entries) in a file 'path.dat', and plot using

```
unset contour
splot [-3:3][-3:3] f(x,y), 'path.dat' with lines
```

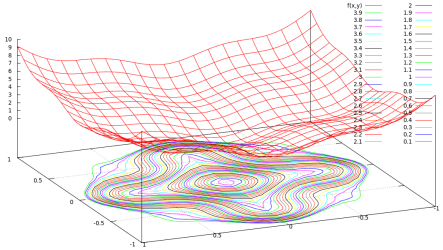
[8] The word "conditioning" generally denotes the ratio of the largest and smallest Eigenvalue of the Hessian.

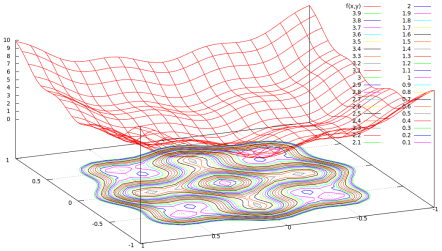
b) Play around with parameters. How does the performance change for higher dimensions, e.g., $n = 100$? How does the performance change with ρ_{ls} (the Wolfe stop criterion)? How does the alternative in step 3 work?

c) Newton step: Modify the algorithm simply by multiplying C^{-1} to the step. How does that work?

(The Newton direction diverges (is undefined) in the concave part of $f_{\text{hole}}(x)$. We're cheating here when always multiplying with C^{-1} to get a good direction.)

4.8.3 Gauss-Newton





In $x \in \mathbb{R}^2$ consider the function

$$f(x) = \phi(x)^\top \phi(x) \ , \quad \phi(x) = \begin{pmatrix} \sin(ax_1) \\ \sin(acx_2) \\ 2x_1 \\ 2cx_2 \end{pmatrix}$$

The function is plotted above for $a = 4$ (left) and $a = 5$ (right, having local minima), and conditioning $c = 1$. The function is non-convex.

Extend your backtracking method implemented in the last week's exercise to a Gauss-Newton method (with constant λ) to solve the unconstrained minimization problem $\min_x f(x)$ for a random start point in $x \in [-1, 1]^2$. Compare the algorithm for $a = 4$ and $a = 5$ and conditioning $c = 3$ with gradient descent.

4.8.4 Robust unconstrained optimization

A 'flattened' variant of the Rosenbrock function is defined as

$$f(x) = \log[1 + (x_2 - x_1^2)^2 + \frac{1}{100}(1 - x_2)^2]$$

and has the minimum at $x^* = (1, 1)$. For reference, the gradient and hessian are

$$g(x) := 1 + (x_2 - x_1^2)^2 + \frac{1}{100}(1 - x_2)^2 \tag{201}$$

$$\partial_{x_1} f(x) = \frac{1}{g(x)}\Big[-4(x_2 - x_1^2)x_1\Big] \tag{202}$$

$$\partial_{x_2} f(x) = \frac{1}{g(x)}\Big[2(x_2 - x_1^2) - \frac{2}{100}(1 - x_2)\Big] \tag{203}$$

$$\partial^2_{x_1} f(x) = -\Big[\partial_{x_1} f(x)\Big]^2 + \frac{1}{g(x)}\Big[8x_1^2 - 4(x_2 - x_1^2)\Big] \tag{204}$$

$$\partial^2_{x_2} f(x) = -\Big[\partial_{x_2} f(x)\Big]^2 + \frac{1}{g(x)}\Big[2 + \frac{2}{100}\Big] \tag{205}$$

$$\partial_{x_1}\partial_{x_2} f(x) = -\Big[\partial_{x_1} f(x)\Big]\Big[\partial_{x_2} f(x)\Big] + \frac{1}{g(x)}\Big[-4x_1\Big] \tag{206}$$

a) Use gnuplot to display the function copy-and-pasting the following lines:

```
set isosamples 50,50
set contour
f(x,y) = log(1+(y-(x**2))**2 + .01*(1-x)**2 ) - 0.01
splot [-3:3][-3:4] f(x,y)
```

(The '-0.01' ensures that you can see the contour at the optimum.) List and discuss at least three properties of the function (at different locations) that may raise problems to naive optimizers.

b) Use $x = (-3, 3)$ as starting point for an optimization algorithm. Try to code an optimization method that uses all ideas mentioned in the lecture. Try to tune it to be efficient on this problem (without cheating, e.g. by choosing a perfect initial stepsize.)

4.8.5 Lagrangian Method of Multipliers

In a previous exercise we defined the "hole function" $f^c_{\text{hole}}(x)$. Assume conditioning $c = 10$ and use the Lagrangian Method of Multipliers to solve on paper the following constrained optimization problem in $2D$:

$$\min_x f^c_{\text{hole}}(x) \quad \text{s.t.} \quad h(x) = 0 \tag{207}$$

$$h(x) = v^\top x - 1 \tag{208}$$

Near the very end, you won't be able to proceed until you have special values for v. Go as far as you can without the need for these values.

4.8.6 Equality Constraint Penalties and Augmented Lagrangian

The squared penalty approach to solving a constrained optimization problem minimizes

$$\min_x \; f(x) + \mu \sum_{i=1}^m h_i(x)^2 \; . \tag{209}$$

The Augmented Lagrangian method adds a Lagrangian term and minimizes

$$\min_x \; f(x) + \mu \sum_{i=1}^m h_i(x)^2 + \sum_{i=1}^m \lambda_i h_i(x) \; . \tag{210}$$

Assume that we first minimize (209) we end up at a minimum $\widehat{x}$.

Now prove that setting $\lambda_i = 2\mu h_i(\widehat{x})$ will, if we assume that the gradients $\nabla f(x)$ and $\nabla h(x)$ are (locally) constant, ensure that the minimum of (210) fulfills the constraints $h(x) = 0$.

4.8.7 Lagrangian and dual function

(Taken roughly from 'Convex Optimization', Ex. 5.1)

Consider the optimization problem

$$\min_x \; x^2 + 1 \qquad \text{s.t.} \quad (x-2)(x-4) \leq 0$$

with variable $x \in \mathbb{R}$.

a) Derive the optimal solution x^* and the optimal value $p^* = f(x^*)$ by hand.

b) Write down the Lagrangian $L(x, \lambda)$. Plot (using gnuplot or so) $L(x, \lambda)$ over x for various values of $\lambda \geq 0$. Verify the lower bound property $\min_x L(x, \lambda) \leq p^*$, where p^* is the optimum value of the primal problem.

c) Derive the dual function $l(\lambda) = \min_x L(x, \lambda)$ and plot it (for $\lambda \geq 0$). Derive the dual optimal solution $\lambda^* = \text{argmax}_\lambda \, l(\lambda)$. Is $\max_\lambda l(\lambda) = p^*$ (strong duality)?

4.8.8 Optimize a constrained problem

Consider the following constrained problem

$$\min_x \sum_{i=1}^n x_i \;\; \text{s.t.} \;\; g(x) \leq 0 \tag{211}$$

$$g(x) = \begin{pmatrix} x^\top x - 1 \\ -x_1 \end{pmatrix} \tag{212}$$

a) First, assume $x \in \mathbb{R}^2$ is 2-dimensional, and draw on paper what the problem looks like and where you expect the optimum.

b) Find the optimum analytically using the Lagrangian. Here, assume that you know apriori that all constraints are active! What are the dual parameters $\lambda = (\lambda_1, \lambda_2)$?

Note: Assuming that you know a priori which constraints are active is a huge assumption! In real problems, this is the actual hard (and combinatorial) problem. More on this later in the lecture.

c) Implement a simple the Log Barrier Method. Tips:

- Initialize $x = (\frac{1}{2}, \frac{1}{2})$ and $\mu = 1$
- First code an inner loop:
  - In each iteration, first compute the gradient of the log-barrier function. Recall that

$$F(x;\mu) = f(x) - \mu \sum_i \log(-g_i(x)) \tag{213}$$

$$\nabla F(x;\mu) = \nabla f - \mu \sum_i (1/g_i(x)) \nabla g_i(x) \tag{214}$$

  - Then perform a backtracking line search along $-\nabla F(x,\mu)$. In particular, backtrack if a step goes beyond the barrier (where $g(x) \not\leq 0$ and $F(x,\mu) = \infty$).
  - Iterate until convergence; let's call the result $x^*(\mu)$. Further, compute $\lambda^*(m) = -(\mu/g_1(x), \mu/g_2(x))$ at convergence.
- Decrease $\mu \leftarrow \mu/2$, recompute $x^*(\mu)$ (with the previous x^* as initialization) and iterate this.

Does x^* and λ^* converge to the expected solution?

Note: The path $x^*(\mu) = \text{argmin}_x F(x;\mu)$ (the optimum in dependence of μ) is called *central path*.

Comment: Solving problems in the real world involves 2 parts:

1) formulating the problem as an optimization problem (conform to a standard optimization problem category) ($\rightarrow$ human)

2) the actual optimization problem ($\rightarrow$ algorithm)

These exercises focus on the first type, which is just as important as the second, as it enables the use of a wider range of solvers. Exercises from Boyd et al http://www.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf:

4.8.9 Network flow problem

Solve Exercise 4.12 (pdf page 193) from Boyd & Vandenberghe, *Convex Optimization.*

4.8.10 Minimum fuel optimal control

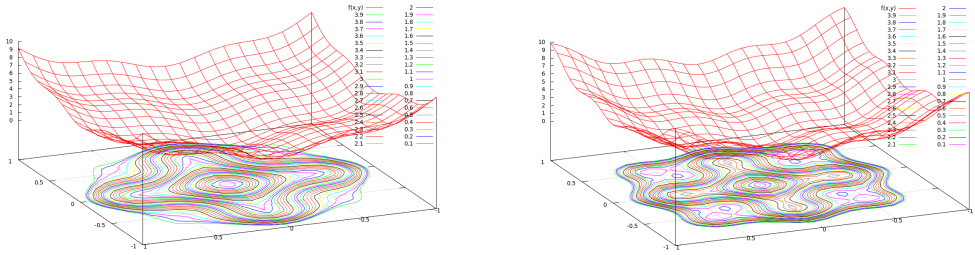
Solve Exercise 4.16 (pdf page 194) from Boyd & Vandenberghe, *Convex Optimization.*

4.8.11 Reformulating an ℓ_1-norm

(This is a subset of Exercise 4.11 (pdf page 193) from Boyd & Vandenberghe.)

Let $x \in \mathbb{R}^n$. The optimization problem is $\min_x \|x - b\|_1$, where the ℓ_1-norm is defined as $\|z\|_1 = \sum_{i=1}^n |z_i|$. Reformulate this optimization problem as a Linear Program.

4.8.12 Restarts of Local Optima



The following function is essentially the Rastrigin function, but written slightly differently. It can be tuned to become uni-modal and is a sum-of-squares problem. For $x \in \mathbb{R}^2$ we define

$$f(x) = \phi(x)^\top \phi(x) \ , \quad \phi(x) = \begin{pmatrix} \sin(ax_1) \\ \sin(acx_2) \\ 2x_1 \\ 2cx_2 \end{pmatrix}$$

The function is plotted above for $a = 4$ (left) and $a = 5$ (right, having local minima), and conditioning $c = 1$. The function is non-convex.

Choose $a = 6$ or larger and implement a random restart method: Repeat initializing $x \sim \mathcal{U}([-2, 2]^2)$ uniformlly, followed by a gradient descent (with backtracking line search and monotone convergence).

Restart the method at least 100 times. Count how often the method converges to which local optimum.

4.8.13 GP-UCB Bayesian Optimization

Find an implementation of Gaussian Processes for your language of choice (e.g. python: scikit-learn, or Sheffield/Gpy; octave/matlab: gpml) and implement GP-UCB global

optimization. Test your implementation with different hyperparameters (Find the best combination of kernel and its parameters in the GP) on the 2D function defined above.

On the webpage you find a starting code to use GP regression in scikit-learn. To install scikit-learn: https://scikit-learn.org/stable/install.html

5 Probabilities & Information

It is beyond the scope of these notes to give a detailed introduction to probability theory. There are excellent books:

- Thomas & Cover
- Bishop
- MacKay

Instead, we first recap very basics of probability theory, that I assume the reader has already seen before. The next section will cover this. Then we focus on specific topics that, in my opinion, deepen the understanding of the basics, such as the relation between optimization and probabilities, log-probabilities & energies, maxEntropy and maxLikelihood, minimal description length and learning.

5.1 Basics

First, in case you wonder about justifications of the use of (Bayesian) probabilities versus fuzzy sets or alike, here some pointers to look up: 1) Cox's theorem, which derives from basic assumptiond about "rationality and consistency" the standard probability axioms; 2) t-norms, which generalize probability and fuzzy calculus; and 3) read about objective vs. subjecte Bayesian probability.

5.1.1 Axioms, definitions, Bayes rule

Definition 5.1 (set-theoretic axioms of probabilities)**.**

- An experiment can have multiple outcomes; we call the set of possible outcomes **sample space** or **domain** S
- A mapping $P : A \subseteq S \mapsto [0,1]$, that maps any subset $A \subseteq S$ to a real number, is called **probability measure** on S iff
  - $P(A) \geq 0$ for any $A \subseteq S$ (non-negativity)
  - $P(\bigcup_i A_i) = \sum_i P(A_i)$ if $A_i \cap A_j = \emptyset$ (additivity)

  - $P(S) = 1$ (normalization)

- Implications are:
  - $0 \leq P(A) \leq 1$
  - $P(\emptyset) = 0$
  - $A \subseteq B \Rightarrow P(A) \leq P(B)$
  - $P(A \cup B) = P(A) + P(B) - P(A \cap B)$
  - $P(S \setminus A) = 1 - P(A)$

Formally, a **random variable** X is a mapping $X : S \to \Omega$ from a measureable space S (that is, a sample space S that has a probability measure P) to another sample space Ω, which I typically call the **domain** $\mathrm{dom}(X)$ of the random variable. Thereby, the mapping $X : S \to \Omega$ now also defines a probability measure over the domain Ω:

$$P(B \subseteq \Omega) = P(\{s : X(s) \in B\}) \tag{215}$$

In practise we just use the following notations:

Definition 5.2 (Random Variable)**.**

- Let X be a random variable with discrete domain $\mathrm{dom}(X) = \Omega$
- $P(X{=}x) \in \mathbb{R}$ denotes the specific probability that $X = x$ for some $x \in \Omega$
- $P(X)$ denotes the **probability distribution** (function over Ω)
- $\forall_{x\in\Omega} :\ 0 \leq P(X{=}x) \leq 1$
- $\sum_{x\in\Omega} P(X{=}x) = 1$
- We often use the short hand $\sum_X P(X)\ \cdots = \sum_{x\in\mathrm{dom}(X)} P(X = x)\ \cdots$ when summing over possible values of a RV

If we have two or more random variables, we have

Definition 5.3 (Joint, marginal, conditional, independence, Bayes' Theorem)**.**

- We denote the **joint** distribution of two RVs as $P(X, Y)$
- The **marginal** is defined as $P(X) = \sum_Y P(X, Y)$
- The **conditional** is defined as $P(X|Y) = \frac{P(X,Y)}{P(Y)}$, which fulfils $\forall_Y :\ \sum_X P(X|Y) = 1$.
- X is **independent** of Y iff $P(X, Y) = P(X)\ P(Y)$, or equivalently, $P(X|Y) = P(X)$.

- The definition of a conditional implies the **product rule**

$$P(X,Y) = P(X|Y)\ P(Y) = P(Y|X)\ P(X) \tag{216}$$

and **Bayes' Theorem**

$$P(X|Y) = \frac{P(Y|X)\ P(X)}{P(Y)} \tag{217}$$

The individual terms in Bayes' Theorem are typically given names:

$$\text{posterior} \ = \ \frac{\text{likelihood} \ \cdot \ \text{prior}}{\text{normalization}} \tag{218}$$

(Sometimes, the normalization is also called *evidence*.)

- X is *conditionally independent* of Y given Z iff $P(X|Y,Z) = P(X|Z)$ or $P(X,Y|Z) = P(X|Z)\ P(Y|Z)$

5.1.2 Standard discrete distributions

	RV	parameter	distribution
Bernoulli	$x \in \{0,1\}$	$\mu \in [0,1]$	$\text{Bern}(x \,\|\, \mu) = \mu^x(1-\mu)^{1-x}$
Beta	$\mu \in [0,1]$	$\alpha, \beta \in \mathbb{R}^+$	$\text{Beta}(\mu \,\|\, a,b) = \frac{1}{B(a,b)}\ \mu^{a-1}(1-\mu)^{b-1}$
Multinomial	$x \in \{1,..,K\}$	$\mu \in [0,1]^K,\ \|\mu\|_1 = 1$	$P(x=k \,\|\, \mu) = \mu_k$
Dirichlet	$\mu \in [0,1]^K,\ \|\mu\|_1 = 1$	$\alpha_1,..,\alpha_K \in \mathbb{R}^+$	$\text{Dir}(\mu \,\|\, \alpha) \propto \prod_{k=1}^K \mu_k^{\alpha_k - 1}$

Clearly, the Multinomial is a generalization of the Bernoulli, as the Dirichlet is of the Beta. The mean of the Dirichlet is $\langle \mu_i \rangle = \frac{\alpha_i}{\sum_j \alpha_j}$, its mode is $\mu_i^* = \frac{\alpha_i - 1}{\sum_j \alpha_j - K}$. The **mode** of a distribution $p(x)$ is defined as $\text{argmax}_x\, p(x)$.

5.1.3 Conjugate distributions

Definition 5.4 (Conjugacy)**.** Let $p(D|x)$ be a likelihood conditional on a RV x. A family $\mathcal{C}$ of distributions (i.e., $\mathcal{C}$ is a space of distributions, like the space of all Beta distributions) is called **conjugate** to the likelihood function $p(D|x)$ iff

$$p(x) \in \mathcal{C} \quad \Rightarrow \quad p(x|D) = \frac{p(D|x)\ p(x)}{p(D)} \in \mathcal{C}\ . \tag{219}$$

The standard conjugates you should know:

RV	likelihood	conjugate
μ	Binomial $\mathrm{Bin}(D \mid \mu)$	Beta $\mathrm{Beta}(\mu \mid a, b)$
μ	Multinomial $\mathrm{Mult}(D \mid \mu)$	Dirichlet $\mathrm{Dir}(\mu \mid \alpha)$
μ	Gauss $\mathcal{N}(x \mid \mu, \Sigma)$	Gauss $\mathcal{N}(\mu \mid \mu_0, A)$
λ	1D Gauss $\mathcal{N}(x \mid \mu, \lambda^{-1})$	Gamma $\mathrm{Gam}(\lambda \mid a, b)$
Λ	nD Gauss $\mathcal{N}(x \mid \mu, \Lambda^{-1})$	Wishart $\mathrm{Wish}(\Lambda \mid W, \nu)$
(μ, Λ)	nD Gauss $\mathcal{N}(x \mid \mu, \Lambda^{-1})$	Gauss-Wishart $\mathcal{N}(\mu \mid \mu_0, (\beta\Lambda)^{-1})\ \mathrm{Wish}(\Lambda \mid W, \nu)$

5.1.4 Distributions over continuous domain

Definition 5.5. Let x be a continuous RV. The **probability density function (pdf)** $p(x) \in [0, \infty)$ defines the probability

$$P(a \leq x \leq b) = \int_a^b p(x)\ dx\ \in [0, 1] \tag{220}$$

The **cumulative probability distribution** $F(y) = P(x \leq y) = \int_{-\infty}^{y} dx\ p(x) \in [0, 1]$ is the cumulative integral with $\lim_{y \to \infty} F(y) = 1$

However, I and most others say **probability distribution** to refer to probability density function.

One comment about integrals. If $p(x)$ is a probability density function and $f(x)$ some arbitrary function, typically one writes

$$\int_x f(x)\ p(x)\ dx\ , \tag{221}$$

where dx denotes the (Borel) measure we integrate over. However, some authors (correctly) think of a distribution $p(x)$ as being a measure over the space $\mathrm{dom}(x)$ (instead of just a function). So the above notation is actually "double" w.r.t. the measures. So they might (also correctly) write

$$\int_x p(x)\ f(x)\ , \tag{222}$$

and take care that there is exactly one measure to the right of the integral.

5.1.5 Gaussian

Definition 5.6. We define an n-dim Gaussian in *normal form* as

$$\mathcal{N}(x \mid \mu, \Sigma) = \frac{1}{|\, 2\pi\Sigma \,|^{1/2}}\ \exp\{-\frac{1}{2}(x-\mu)^\top\, \Sigma^{-1}\ (x-\mu)\} \tag{223}$$

with **mean** μ and **covariance** matrix Σ. In *canonical form* we define

$$\mathcal{N}[x\,|\,a,A] = \frac{\exp\{-\frac{1}{2}a^\top A^{-1}a\}}{|\,2\pi A^{-1}\,|^{1/2}}\ \exp\{-\frac{1}{2}x^\top\ A\ x + x^\top a\} \tag{224}$$

with **precision** matrix $A = \Sigma^{-1}$ and coefficient $a = \Sigma^{-1}\mu$ (and mean $\mu = A^{-1}a$).

Gaussians are used all over—below we explain in what sense they are the probabilistic analogue to a parabola (or a 2nd-order Taylor expansions). The most important properties are:

- Symmetry: $\mathcal{N}(x\,|\,a,A) = \mathcal{N}(a\,|\,x,A) = \mathcal{N}(x-a\,|\,0,A)$
- Product:
  $\mathcal{N}(x\,|\,a,A)\ \mathcal{N}(x\,|\,b,B) = \mathcal{N}[x\,|\,A^{-1}a + B^{-1}b, A^{-1}+B^{-1}]\ \mathcal{N}(a\,|\,b,A+B)$
  $\mathcal{N}[x\,|\,a,A]\ \mathcal{N}[x\,|\,b,B] = \mathcal{N}[x\,|\,a+b, A+B]\ \mathcal{N}(A^{-1}a\,|\,B^{-1}b, A^{-1}+B^{-1})$
- "Propagation":
  $\int_y \mathcal{N}(x\,|\,a+Fy,A)\ \mathcal{N}(y\,|\,b,B)\ dy = \mathcal{N}(x\,|\,a+Fb, A+FBF^\top)$
- Transformation:
  $\mathcal{N}(Fx+f\,|\,a,A) = \frac{1}{|\,F\,|}\ \mathcal{N}(x\,|\ F^{-1}(a-f),\ F^{-1}AF^{-\top})$
- Marginal & conditional:
  $$\mathcal{N}\left(\begin{array}{c|cc} x & a & A & C \\ y & b & C^\top & B \end{array}\right) = \mathcal{N}(x\,|\,a,A)\cdot\mathcal{N}(y\,|\,b+C^\top A^{-1}(x-a),\ B-C^\top A^{-1}C)$$

More Gaussian identities are found at http://ipvs.informatik.uni-stuttgart.de/mlr/marc/notes/gaussians.pdf

Example 5.1 (ML estimator of the mean of a Gaussian)**.** Assume we have data $D = \{x_1, .., x_n\}$, each $x_i \in \mathbb{R}^n$, with likelihood

$$P(D\,|\,\mu,\Sigma) = \textstyle\prod_i \mathcal{N}(x_i\,|\,\mu,\Sigma) \tag{225}$$

$$\operatorname*{argmax}_\mu\ P(D\,|\,\mu,\Sigma) = \frac{1}{n}\sum_{i=1}^n x_i \tag{226}$$

$$\operatorname*{argmax}_\Sigma\ P(D\,|\,\mu,\Sigma) = \frac{1}{n}\sum_{i=1}^n (x_i-\mu)(x_i-\mu)^\top \tag{227}$$

Assume we are initially uncertain about μ (but know Σ). We can express this uncertainty using again a Gaussian $\mathcal{N}[\mu\,|\,a,A]$. Given data we have

$$P(\mu\,|\,D) \propto P(D\,|\,\mu,\Sigma)\ P(\mu) = \textstyle\prod_i \mathcal{N}(x_i\,|\,\mu,\Sigma)\ \mathcal{N}[\mu\,|\,a,A] \tag{228}$$

$$= \textstyle\prod_i \mathcal{N}[\mu\,|\,\Sigma^{-1}x_i, \Sigma^{-1}]\ \mathcal{N}[\mu\,|\,a,A] \propto \mathcal{N}[\mu\,|\,\Sigma^{-1}\textstyle\sum_i x_i,\ n\Sigma^{-1}+A] \tag{229}$$

Note: in the limit $A \to 0$ (uninformative prior) this becomes

$$P(\mu \,|\, D) = \mathcal{N}(\mu \,|\, \frac{1}{n}\sum_i x_i, \frac{1}{n}\Sigma) \tag{230}$$

which is consistent with the Maximum Likelihood estimator

5.1.6 "Particle distribution"

Usually, "particles" are not listed as standard continuous distribution. However I think they should be. They're heavily used in several contexts, especially as approximating other distributions in Monte Carlo methods and particle filters.

Definition 5.7 (Dirac or δ-distribution)**.** In *distribution theory* it is proper to define a distribution $\delta(x)$ that is the derivative of the Heavyside step function $H(x)$,

$$\delta(x) = \frac{\partial}{\partial x} H(x) \ , \quad H(x) = [x \geq 0] \ . \tag{231}$$

It is akward to think of $\delta(x)$ as a normal function, as it'd be "infinite" at zero. But at least we understand that is has the properties

$$\delta(x) = 0 \text{ everywhere except at } x = 0 \ , \quad \int \delta(x)\, dx = 1 \ . \tag{232}$$

I sometimes call the Dirac distribution also a **point particle**: it has all its unit "mass" concentrated at zero.

Definition 5.8 (Particle Distribution)**.** We define a **particle distribution** $q(x)$ as a **mixture** of Diracs,

$$q(x) := \sum_{i=1}^{N} w_i \ \delta(x - x_i) \ , \tag{233}$$

which is parameterized by the number N, the locations $\{x_i\}_{i=1}^N$, $x_i \in \mathbb{R}^n$, and the normalized weights $\{w_i\}_{i=1}^N$, $w_i \in \mathbb{R}$, $\|w\|_1 = 1$ of the N particles.

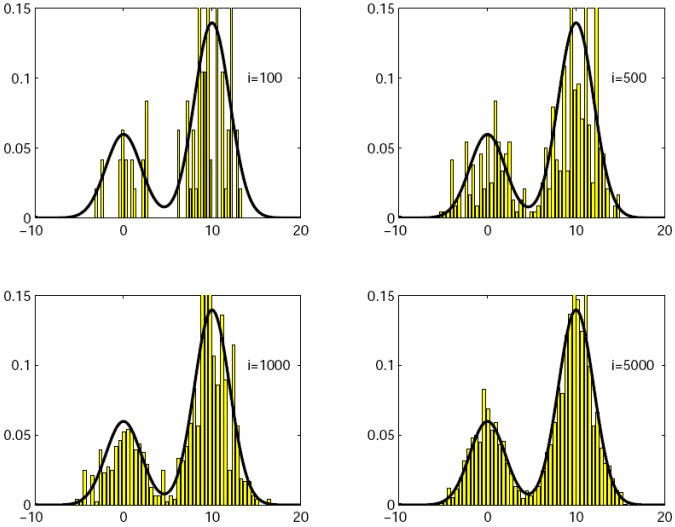
We say that a particle distribution $q(x)$ approximates another distribution $p(x)$ iff for any (smooth) f

$$\langle f(x) \rangle_p = \int_x f(x) p(x) dx \ \approx \ \textstyle\sum_{i=1}^N w_i f(x_i) \tag{234}$$

Note the generality of this statement! f could be anything, it could be any features of the variable x, like coordinates of x, or squares, or anything. So basically this statement

says, whatever you might like to estimate about p, you can approximate it based on the particles q.

Computing particle approximations of comples (non-analytical, non-tracktable) distributions p is a core challenge in many fields. The true p could for instance be a distributions over games (action sequences). The approximation q could for instance be samples generated with Monte Carlo Tree Search (MCTS). The tutorial *An Introduction to MCMC for Machine Learning* www.cs.ubc.ca/~nando/papers/mlintro.pdf gives an excellent introduction. Here are some illustrations of what it means to approximate some p by particles q, taken from this tutorial. The black line is p, histograms illustrate the particles q by showing how many of (uniformly weighted) particles fall into a bin:



(from de Freitas et al.)

5.2 Between probabilities and optimization: neg-log-probabilities, exp-neg-energies, exponential family, Gibbs and Boltzmann

There is a natural relation between probabilities and "energy" (or "error"). Namely, if $p(x)$ denotes a probability for every possible value of x, and $E(x)$ denotes an energy for state x—or an error one assigns to choosing x—then a natural relation is

$$p(x) = e^{-E(x)} \ , \quad E(x) = -\log p(x) \ . \tag{235}$$

Why is that? First, outside the context of physics it is perfectly fair to just define axiomatically an energy $E(x)$ as neg-log-probability. But let me try to give some more arguments for why this is a useful definition.

Let assume we have $p(x)$. We want to find a quantity, let's call it *error* $E(x)$, which is a function of $p(x)$. Intuitively, if a certain value x_1 is more likely than another,

$p(x_1) > p(x_2)$, then picking x_1 should imply less error, $E(x_1) < E(x_2)$ (Axiom 1). Further, when we have two independent random variables x and y, **probabilities are multiplicative**, $p(x,y) = p(x)p(y)$. We require axiomatically that **error is additive**, $E(x,y) = E(x) + E(y)$. From both follows that E needs to be some logarithm of p!

The same argument, now more talking about *energy*: Assume we have two independent (physical) systems x and y. $p(x,y) = p(x)p(y)$ is the probability to find them in certain states. We axiomatically require that **energy is additive**, $E(x,y) = E(x) + E(y)$. Again, E needs to be some logarithm of p. In the context of physics, what could be questioned is "why is $p(x)$ a function of $E(x)$ in the first place?". Well, that is much harder to explain and really is a question about statistical physics. Wikipedia under keywords "Maxwell-Boltzmann statistics" and "Derivation from microcanonical ensemble" gives an answer. Essentially the argument is a follows: Given many many molecules in a gas, each of which can have a different energy e_i. The total energy $E = \sum_{i=1}^{n} e_i$ must be conserved. What is the distribution over energy levels that has the most microstates? The answer is the Boltzmann distribution. (And why do we, in nature, find energy distributions that have the most microstates? Because these are most likely.)

Bottom line is: $p(x) = e^{-E(x)}$, probabilities are multiplicative, energies or errors additive.

Let me state some fact just to underline how useful this way of thinking is:

- Given an energy function $E(x)$, its **Boltzmann distribution** is defined as

$$p(x) = e^{-E(x)}. \tag{236}$$

  This is sometimes also called Gibbs distribution.

- In machine learning, when data D is given and we have some model β, we typically try to maximize the likelihood $p(D|\beta)$. This is equivalent to minimizing the neg-log-likelihood

$$L(\beta) = -\log p(D|\beta) \ . \tag{237}$$

  This neg-log-likelihood is a typical measure for *error* of the model. And this error is additive w.r.t. the data, whereas the likelihood is multiplicative, fitting perfectly to the above discussion.

- The Gaussian distribution $p(x) \propto \exp\{-\frac{1}{2}\|x-\mu\|^2/\sigma^2\}$ is related to the error $E(x) = \frac{1}{2}\|x-\mu\|^2/\sigma^2$, which is nothing but the squared error with the precision matrix as metric. That's why squared error measures (classical regression) and Gaussian distributions (e.g., Bayesian Ridge regression) are directly related.

  A Gaussian is the probabilistic analoque to a parabola.

- The exponential family is defined as

$$p(x|\beta) = h(x)g(\beta)\exp\{\beta^\top \phi(x)\} \tag{238}$$

Often $h(x) = 1$, so let's neglect this for now. The key point is that the energy is linear in the features $\phi(x)$. This is exactly how discriminative functions (for classification in Machine learning) are typically formulated.

In the continuous case, the features $\phi(x)$ are often chosen as basis polynomials—just as in polynomial regression. Then, β are the coefficients of the energy polynomial and the exponential family is just the probabilistic analogue to the space of polynomials.

- When we have many variables $x_1, .., x_n$, the structure of a cost function over these variables can often be expressed as being additive in terms: $f(x_1, .., x_n) = \sum_i \phi_i(x_{\partial i})$ where ∂i denotes the ith group of variables. The respective Boltzmann distribution is a **factor graph** $p(x_1, .., x_n) \propto \prod_i f_i(x_{\partial i}) = \exp\{\sum_i \beta_i \phi_i(x_{\partial i})$ where ∂i denotes the

  So, factor graphs are the probabilistic analoque to additive functions.

- $-\log p(x)$ is also the "optimal" coding length you should assign to a symbol x.

  Entropy is expected error: $H[p] = \sum_x -p(x) \log p(x) = \langle -\log p(x) \rangle_{p(x)}$, where p itself it used to take the expectation.

  Assume you use a "wrong" distribution $q(x)$ to decide on the coding length of symbols drawn from $p(x)$. The expected length of an encoding is $\int_x p(x)[-\log q(x)] \geq H(p)$.

  The **Kullback-Leibler divergence** is the difference:

$$D\big(p \,\|\, q\big) = \int_x p(x) \log \frac{p(x)}{q(x)} \geq 0 \tag{239}$$

  Proof of inequality, using the Jenson inequality:

$$-\int_x p(x) \log \frac{q(x)}{p(x)} \geq -\log \int_x p(x) \frac{q(x)}{p(x)} = 0 \tag{240}$$

So, my message is that probabilities and error measures are naturally related. However, in the first case we typically do inference, in the second we optimize. Let's discuss the relation between inference and optimization a bit more. For instance, given data D and parameters β, we may define

Definition 5.9 (ML, MAP, and Bayes estimate)**.** Given data D and a parameteric model $p(D|\beta)$, we define

- **Maximum likelihood (ML) parameter estimate:**
  $\beta^{\text{ML}} := \text{argmax}_\beta \, P(D|\beta)$
- **Maximum a posteriori (MAP) parameter estimate:**
  $\beta^{\text{MAP}} = \text{argmax}_\beta \, P(\beta|D)$

- **Bayesian parameter estimate:**
  $P(\beta|D) \propto P(D|\beta)\ P(\beta)$
  used for **Bayesian prediction:** $P(\text{prediction}|D) = \int_\beta P(\text{prediction}|\beta)\ P(\beta|D)$

Both, the MAP and the ML estimates are really just optimization problems.

The Bayesian parameter estimate $P(\beta|D)$, which can then be used to do fully Bayesian prediction, is in principle different. However, in practise also here optimization is a core tool for estimating such distributions if they cannot be given analytically. This is described next.

5.3 Information, Entropie & Kullback-Leibler

Consider the following problem. We have data drawn i.i.d. from $p(x)$ where $x \in X$ in some discrete space X. Let's call every x a *word*. The problem is to find a mapping from words to *codes*, e.g. binary codes $c: X \to \{0,1\}^*$. The optimal solution is in principle simple: Sort all possible words in a list, ordered by $p(x)$ with more likely words going first; write all possible binary codes in another list, with increasing code lengths. Match the two lists, and this is the optimal encoding.

Let's try to get a more analytical grip of this: Let $l(x) = |c(x)|$ be the actual code length assigned to word x, which is an integer value. Let's define

$$q(x) = \frac{1}{Z}\ 2^{-l(x)} \tag{241}$$

with the normalization constrant $Z = \sum_x 2^{-l(x)}$. Then we have

$$\sum_{x\in X} p(x)[-\log_2 q(x)] = -\sum_x p(x) \log 2^{-l(x)} + \sum_x p(x) \log Z \tag{242}$$

$$= \sum_x p(x) l(x) + \log Z\ . \tag{243}$$

What about $\log Z$? Let $l^{-1}(s)$ be the set of words that have been assigned codes of length l. There can only be a limited number of words encoded with a given length. For instance, $|L^{-1}(1)|$ must not be greater than 2, $|L^{-1}(2)|$ must not be greater than 4, and $|l^{-1}(s)|$ must not be greater than 2^l. We have

$$\forall_s: \ \sum_{x\in X} [l(x) = s] \le 2^s \tag{244}$$

$$\forall_s: \ \sum_{x\in X} [l(x) = s] 2^{-s} \le 1 \tag{245}$$

$$\forall_s: \ \sum_{x\in X} 2^{-l(x)} \le 1 \tag{246}$$

However, this way of thinking is ok for separated codes. If such codes would be in a continuous stream of bits you'd never know where a code starts or ends. Prefix codes fix this problem by defining a code tree with leaves that clearly define when a code ends. For prefix codes it similarly holds

$$Z = \sum_{x \in X} 2^{-l(x)} \leq 1 \ , \tag{247}$$

which is called *Kraft's inequality*. That finally gives

$$\sum_{x \in X} p(x)[-\log_2 q(x)] \leq \sum_x p(x) l(x) \tag{248}$$

5.4 The Laplace approximation: A 2nd-order Taylor of $\log p$

Assume we want to estimate some $q(x)$ we cannot express analytically. E.g., $q(x) = p(x|D) \propto P(D|x)p(x)$ for some awkward likelihood function $p(D|x)$. An example from robotics is: x is stochastically controlled path of a robot. $p(x)$ is a prior distribution over paths that includes how the robot can actually move and some Gaussian prior (squared costs!) over controlls. If the robot is "linear", $p(x)$ can be expressed nicely and analytically; it if it non-linear, expressing $p(x)$ is already hard. However, $p(D|x)$ might indicate that we do *not* see collisions on the path—but collisions are a horrible function, usually computed by some black-box collision detection packages that computes distances between convex meshes, perhaps giving gradients but certainly not some analytic function. So $q(x)$ can clearly not be expressed analytically.

One way to approximate $q(x)$ is the Laplace approximation

Definition 5.10 (Laplace approximation)**.** Given a smooth distribution $q(x)$, we define its Laplace approximation as

$$\tilde{q}(x) = \exp\{-\tilde{E}(x)\} \ , \tag{249}$$

where $\tilde{E}(x)$ is the 2nd-order Taylor expansion

$$\tilde{E}(x) = E(x^*) + \frac{1}{2}(x - x^*)^\top \nabla^2 E(x^*)(x - x^*) \tag{250}$$

of the energy $E(x) = -\log q(x)$ at the mode

$$x^* = \operatorname*{argmin}_x E(x) = \operatorname*{argmax}_x q(x) \ . \tag{251}$$

First, we observe that the Laplace approximation is a Gaussian, because its energy is a parabola. Further, notice that in the Taylor expansion we skipped the linear term. That's because we are at the mode x^* where $\nabla E(x^*) = 0$.

The Laplace approximation is the probabilistic analoque of a local second-order approximation of a function, just as we used it in Newton methods. However, it is defined to be taken specifically at the mode of the distribution.

Now, computing x^* is a classical optimization problem $x^* = \text{argmin}_x E(x)$ which one might ideally solve using Newton methods. These Newton methods anyway compute the local Hessian of $E(x)$ in every step—at the optimum we therefore have the Hessian already, which is then the precision matrix of our Gaussian.

The Laplace approximation is nice, very efficient to use, e.g., in the context of optimal control and robotics. While we can use the expressive power of probability theory to formalize the problem, the Laplace approximation brings us computationally back to efficient optimization methods.

5.5 Variational Inference

Another reduction of inference to optimization is variational inference.

Definition 5.11 (variational inference)**.** Given a distribution $p(x)$, and a parameterized family of distributions $q(x|\beta)$, the variational approximation of $p(x)$ is defined as

$$\underset{q}{\text{argmin}}\, D\big(q \,\|\, p\big) \tag{252}$$

5.6 The Fisher information metric: 2nd-order Taylor of the KLD

Recall our notion of steepest descent—it depends on the metric in the space!

Consider the space of probability distributions $p(x;\beta)$ with parameters β. We think of every $p(x;\beta)$ as a point in the space and wonder what metric is useful to compare two points $p(x;\beta_1)$ and $p(x;\beta_2)$. Let's take the KLD

TODO

: Let $p \in \Lambda^X$, that is, p is a probability distribution over the space X. Further, let $\theta \in \mathbb{R}^n$ and $\theta \mapsto p(\theta)$ is some parameterization of the probability distribution. Then the derivative $d_\theta p(\theta) \in T_p\Lambda^X$ is a vector in the tangent space of Λ^X. Now, for such vectors, for tangent vectors of the space of probability distributions, there is a generic metric, the **Fisher metric**: [TODO: move to 'probabilities' section]

5.7 Examples and Exercises

Note: These exercises are for 'extra credits'. We'll discuss them on Thu, 21th Jan.

5.7.1 Maximum Entropy and Maximum Likelihood

(These are taken from MacKay's book *Information Theory...*, Exercise 22.12 & .13)

a) Assume that a random variable x with discrete domain $\text{dom}(x) = \mathcal{X}$ comes from a probability distribution of the form

$$P(x \,|\, w) = \frac{1}{Z(w)} \exp\Big[\sum_{k=1}^{d} w_k f_k(x)\Big] \ ,$$

where the functions $f_k(x)$ are given, and the parameters $w \in \mathbb{R}^d$ are not known. A data set $D = \{x_i\}_{i=1}^n$ of n points x is supplied. Show by differentiating the log likelihood $\log P(D|w) = \sum_{i=1}^n \log P(x_i|w)$ that the maximum-likelihood parameters $w^* = \text{argmax}_w \log P(D|w)$ satisfy

$$\sum_{x\in\mathcal{X}} P(x \,|\, w^*)\ f_k(x) = \frac{1}{n} \sum_{i=1}^{n} f_k(x_i)$$

where the left-hand sum is over all x, and the right-hand sum is over the data points. A shorthand for this result is that each function-average under the fitted model must equal the function-average found in the data:

$$\langle f_k \rangle_{P(x \,|\, w^*)} = \langle f_k \rangle_D$$

b) When confronted by a probability distribution $P(x)$ about which only a few facts are known, the maximum entropy principle (MaxEnt) offers a rule for choosing a distribution that satisfies those constraints. According to MaxEnt, you should select the $P(x)$ that maximizes the entropy

$$H(P) = -\sum_x P(x) \log P(x)$$

subject to the constraints. Assuming the constraints assert that the averages of certain functions $f_k(x)$ are known, i.e.,

$$\langle f_k \rangle_{P(x)} = F_k \ ,$$

show, by introducing Lagrange multipliers (one for each constraint, including normalization), that the maximum-entropy distribution has the form

$$P_{\text{MaxEnt}}(x) = \frac{1}{Z} \exp\Big[\sum_k w_k\ f_k(x)\Big]$$

where the parameters Z and w_k are set such that the constraints are satisfied. And hence the maximum entropy method gives identical results to maximum likelihood fitting of an exponential-family model.

Note: The exercise will take place on Tue, 2nd Feb. Hung will also prepare how much 'votes' you collected in the exercises.

5.7.2 Maximum likelihood and KL-divergence

Assume we have a very large data set $D = \{x_i\}_{i=1}^n$ of samples $x_i \sim q(x)$ from some data distribution $q(x)$. Using this data set we can approximate any expectation

$$\langle f \rangle_q = \sum_x q(x) f(x) \approx \sum_{i=1}^n f(x_i) \ .$$

Assume we have a parameteric family of distributions $p(x|\beta)$ and would find the Maximum Likelihood (ML) parameter $\beta^* = \text{argmax}_\beta \, p(D|\beta)$. Express this ML problem as a KL-divergence minimization.

5.7.3 Laplace Approximation

In the context of so-called "Gaussian Process Classification" the following problem arises (we neglect dependence on x here): We have a real-valued RV $f \in \mathbb{R}$ with prior $P(f) = \mathcal{N}(f \,|\, \mu, \sigma^2)$. Further we have a Boolean RV $y \in \{0, 1\}$ with conditional probability

$$P(y{=}1 \,|\, f) = \sigma(f) = \frac{e^f}{1 + e^f} \ .$$

The function σ is called sigmoid funtion, and f is a discriminative value which predicts $y{=}1$ if it is very positive, and $y{=}0$ if it is very negative. The sigmoid function has the property

$$\frac{\partial}{\partial f} \sigma(f) = \sigma(f) \ (1 - \sigma(f)) \ .$$

Given that we observed $y = 1$ we want to compute the posterior $P(f \,|\, y = 1)$, which cannot be expressed analytically. Provide the Laplace approximation of this posterior.

(Bonus) As an alternative to the sigmoid function $\sigma(f)$, we can use the probit function $\phi(z) = \int_{-\infty}^z \mathcal{N}(x|0, 1) \ dx$ to define the likelihood $P(y{=}1 \,|\, f) = \phi(f)$. Now how can the posterior $P(f \,|\, y{=}1)$ be approximated?

5.7.4 Learning = Compression

In a very abstract sense, learning means to model the distribution $p(x)$ for given data $D = \{x_i\}_{i=1}^n$. This is literally the case for unsupervised learning; regression, classification

and graphical model learning could be viewed as specific instances of this where x factores in several random variables, like input and output.

Show in which sense the problem of learning is equivalent to the problem of compression.

5.7.5 A gzip experiment

Get three text files from the Web, approximately equal length, mostly text (no equations or stuff). Two of them should be in English, the third in Frensh. (Alternatively, perhaps, not sure if it'd work, two of them on a very similar topic, the third on a very different.)

How can you use `gzip` (or some other compression tool) to estimate the mutual information between every pair of files? How can you ensure some "normalized" measures which do not depend too much on the absolute lengths of the text? Do it and check whether in fact you find that two texts are similar while the third is different.

(Extra) Lempel-Ziv algorithms (like gzip) need to build a codebook on the fly. How does that fit into the picture?

5.7.6 Maximum Entropy and ML

(These are taken from MacKay's book *Information Theory...*, Exercise 22.12 & .13)

a) Assume that a random variable x with discrete domain $\text{dom}(x) = \mathcal{X}$ comes from a probability distribution of the form

$$P(x \,|\, w) = \frac{1}{Z(w)} \exp\Big[\sum_{k=1}^{d} w_k f_k(x)\Big] \,, \tag{253}$$

where the functions $f_k(x)$ are given, and the parameters $w \in \mathbb{R}^d$ are not known. A data set $D = \{x_i\}_{i=1}^n$ of n points x is supplied. Show by differentiating the log likelihood $\log P(D|w) = \sum_{i=1}^n \log P(x_i|w)$ that the maximum-likelihood parameters $w^* = \text{argmax}_w \log P(D|w)$ satisfy

$$\sum_{x\in\mathcal{X}} P(x \,|\, w^*)\; f_k(x) = \frac{1}{n} \sum_{i=1}^{n} f_k(x_i) \tag{254}$$

where the left-hand sum is over all x, and the right-hand sum is over the data points. A shorthand for this result is that each function-average under the fitted model must equal the function-average found in the data:

$$\langle f_k \rangle_{P(x \,|\, w^*)} = \langle f_k \rangle_D \tag{255}$$

b) When confronted by a probability distribution $P(x)$ about which only a few facts are known, the maximum entropy principle (MaxEnt) offers a rule for choosing a distribution that satisfies those constraints. According to MaxEnt, you should select the $P(x)$ that maximizes the entropy

$$H(P) = -\sum_x P(x) \log P(x) \tag{256}$$

subject to the constraints. Assuming the constraints assert that the averages of certain functions $f_k(x)$ are known, i.e.,

$$\langle f_k \rangle_{P(x)} = F_k \ , \tag{257}$$

show, by introducing Lagrange multipliers (one for each constraint, including normalization), that the maximum-entropy distribution has the form

$$P_{\text{MaxEnt}}(x) = \frac{1}{Z} \ \exp\Big[\sum_k w_k \ f_k(x)\Big] \tag{258}$$

where the parameters Z and w_k are set such that the constraints are satisfied. And hence the maximum entropy method gives identical results to maximum likelihood fitting of an exponential-family model.

A Gaussian identities

Definitions
We define a Gaussian over x with mean a and covariance matrix A as the function

$$\mathcal{N}(x \,|\, a, A) = \frac{1}{|2\pi A|^{1/2}} \ \exp\{-\frac{1}{2}(x\text{-}a)^\top \ A^{-1} \ (x\text{-}a)\} \tag{259}$$

with property $N(x \,|\, a, A) = N(a|\, x, A)$. We also define the canonical form with precision matrix A as

$$\mathcal{N}[x \,|\, a, A] = \frac{\exp\{-\frac{1}{2}a^\top A^{-1} a\}}{|2\pi A^{-1}|^{1/2}} \ \exp\{-\frac{1}{2}x^\top \ A \ x + x^\top a\} \tag{260}$$

with properties

$$\mathcal{N}[x \,|\, a, A] = \mathcal{N}(x \,|\, A^{-1}a, A^{-1}) \tag{261}$$

$$\mathcal{N}(x \,|\, a, A) = \mathcal{N}[x \,|\, A^{-1}a, A^{-1}] \ . \tag{262}$$

Non-normalized Gaussian

$$\overline{\mathcal{N}}(x, a, A) = |2\pi A|^{1/2} \ \mathcal{N}(x|a, A) \tag{263}$$

$$= \exp\{-\frac{1}{2}(x\text{-}a)^\top \ A^{-1} \ (x\text{-}a)\} \tag{264}$$

Matrices [matrix cookbook: http://www.imm.dtu.dk/pubdb/views/edoc_download.php/3274/pdf/imm3274.pdf]

$$(A^{-1} + B^{-1})^{-1} = A\ (A+B)^{-1}\ B = B\ (A+B)^{-1}\ A \tag{265}$$
$$(A^{-1} - B^{-1})^{-1} = A\ (B-A)^{-1}\ B \tag{266}$$
$$\partial_x |A_x| = |A_x|\ \mathrm{tr}(A_x^{-1}\ \partial_x A_x) \tag{267}$$
$$\partial_x A_x^{-1} = -A_x^{-1}\ (\partial_x A_x)\ A_x^{-1} \tag{268}$$
$$(A + UBV)^{-1} = A^{-1} - A^{-1}U(B^{-1} + VA^{-1}U)^{-1}VA^{-1} \tag{269}$$
$$(A^{-1} + B^{-1})^{-1} = A - A(B + A)^{-1}A \tag{270}$$
$$(A + J^\top BJ)^{-1}J^\top B = A^{-1}J^\top(B^{-1} + JA^{-1}J^\top)^{-1} \tag{271}$$
$$(A + J^\top BJ)^{-1}A = \mathbf{I} - (A + J^\top BJ)^{-1}J^\top BJ \tag{272}$$

(269)=Woodbury; (271,272) holds for pos def A and B

Derivatives

$$\partial_x \mathcal{N}(x|a, A) = \mathcal{N}(x|a, A)\ (-h^\top)\ , \quad h := A^{-1}(x-a) \tag{273}$$
$$\partial_\theta \mathcal{N}(x|a, A) = \mathcal{N}(x|a, A)\ \cdot$$
$$\Big[-h^\top(\partial_\theta x) + h^\top(\partial_\theta a) - \frac{1}{2}\mathrm{tr}(A^{-1}\ \partial_\theta A) + \frac{1}{2}h^\top(\partial_\theta A)h \Big] \tag{274}$$
$$\partial_\theta \mathcal{N}[x|a, A] = \mathcal{N}[x|a, A]\ \Big[-\frac{1}{2}x^\top\partial_\theta A x + \frac{1}{2}a^\top A^{-1}\partial_\theta A A^{-1}a$$
$$+ x^\top\partial_\theta a - a^\top A^{-1}\partial_\theta a + \frac{1}{2}\mathrm{tr}(\partial_\theta A A^{-1}) \Big] \tag{275}$$
$$\partial_\theta \overline{\mathcal{N}}_x(a, A) = \overline{\mathcal{N}}_x(a, A)\ \cdot$$
$$\Big[h^\top(\partial_\theta x) + h^\top(\partial_\theta a) + \frac{1}{2}h^\top(\partial_\theta A)h \Big] \tag{276}$$

Product
The product of two Gaussians can be expressed as

$$\mathcal{N}(x\,|\,a, A)\ \mathcal{N}(x\,|\,b, B)$$
$$= \mathcal{N}[x\,|\,A^{-1}a + B^{-1}b, A^{-1} + B^{-1}]\ \mathcal{N}(a\,|\,b, A + B)\ , \tag{277}$$
$$= \mathcal{N}(x\,|\,B(A+B)^{-1}a + A(A+B)^{-1}b, A(A+B)^{-1}B)\ \mathcal{N}(a\,|\,b, A + B)\ , \tag{278}$$
$$\mathcal{N}[x\,|\,a, A]\ \mathcal{N}[x\,|\,b, B]$$
$$= \mathcal{N}[x\,|\,a + b, A + B]\ \mathcal{N}(A^{-1}a\,|\,B^{-1}b, A^{-1} + B^{-1}) \tag{279}$$
$$= \mathcal{N}[x|\dots]\ \mathcal{N}[A^{-1}a\,|\,A(A+B)^{-1}b, A(A+B)^{-1}B] \tag{280}$$
$$= \mathcal{N}[x|\dots]\ \mathcal{N}[A^{-1}a\,|\,(1-B(A+B)^{-1})\ b,\ (1-B(A+B)^{-1})\ B]\ , \tag{281}$$
$$\mathcal{N}(x\,|\,a, A)\ \mathcal{N}[x\,|\,b, B]$$

$$= \mathcal{N}[x \,|\, A^{-1}a + b, A^{-1} + B]\ \mathcal{N}(a \,|\, B^{-1}b, A + B^{-1}) \tag{282}$$

$$= \mathcal{N}[x|\dots]\ \mathcal{N}[a \,|\, (1\text{-}B(A^{-1}\text{+}B)^{-1})\ b,\ (1\text{-}B(A^{-1}\text{+}B)^{-1})\ B] \tag{283}$$

Convolution

$$\int_x \mathcal{N}(x \,|\, a, A)\ \mathcal{N}(y - x \,|\, b, B)\ dx = \mathcal{N}(y \,|\, a + b, A + B) \tag{284}$$

Division

$$\mathcal{N}(x|a, A)\ /\ \mathcal{N}(x|b, B) = \mathcal{N}(x|c, C)\ /\ \mathcal{N}(c|b, C + B)$$

$$C^{-1}c = A^{-1}a - B^{-1}b$$

$$C^{-1} = A^{-1} - B^{-1} \tag{285}$$

$$\mathcal{N}[x|a, A]\ /\ \mathcal{N}[x|b, B] \propto \mathcal{N}[x|a - b, A - B] \tag{286}$$

Expectations
Let $x \sim \mathcal{N}(x \,|\, a, A)$,

$$\mathbb{E}\{x\}\, g(x) := \int_x \mathcal{N}(x \,|\, a, A)\ g(x)\ dx \tag{287}$$

$$\mathbb{E}\{x\}\, x = a\ , \quad \mathbb{E}\{x\}\, xx^\top = A + aa^\top \tag{288}$$

$$\mathbb{E}\{x\}\, f + Fx = f + Fa \tag{289}$$

$$\mathbb{E}\{x\}\, x^\top x = a^\top a + \mathrm{tr}(A) \tag{290}$$

$$\mathbb{E}\{x\}\, (x\text{-}m)^\top R(x\text{-}m) = (a\text{-}m)^\top R(a\text{-}m) + \mathrm{tr}(RA) \tag{291}$$

Transformation Linear transformations imply the following identities,

$$\mathcal{N}(x \,|\, a, A) = \mathcal{N}(x + f \,|\, a + f,\ A)\ , \quad \mathcal{N}(x \,|\, a, A) = |F|\ \mathcal{N}(Fx \,|\, Fa,\ FAF^\top) \tag{292}$$

$$\mathcal{N}(Fx + f \,|\, a, A) = \frac{1}{|F|}\ \mathcal{N}(x \,|\ F^{-1}(a - f),\ F^{-1}AF^{-\top}) \tag{293}$$

$$= \frac{1}{|F|}\ \mathcal{N}[x \,|\ F^\top A^{-1}(a - f),\ F^\top A^{-1}F]\ , \tag{294}$$

$$\mathcal{N}[Fx + f \,|\, a, A] = \frac{1}{|F|}\ \mathcal{N}[x \,|\ F^\top(a - Af),\ F^\top AF]\ . \tag{295}$$

"Propagation" (propagating a message along a coupling, using eqs (277) and (283), respectively)

$$\int_y \mathcal{N}(x \,|\, a + Fy, A)\ \mathcal{N}(y \,|\, b, B)\ dy = \mathcal{N}(x \,|\, a + Fb, A + FBF^\top) \tag{296}$$

$$\int_y \mathcal{N}(x \,|\, a + Fy, A)\ \mathcal{N}[y \,|\, b, B]\ dy = \mathcal{N}[x \,|\, (F^{-\top}\text{-}K)(b + BF^{-1}a),\ (F^{-\top}\text{-}K)BF^{-1}]\ , \tag{297}$$

$$K = F^{-\top}B(F^{-\top}A^{-1}F^{-1}+B)^{-1} \tag{298}$$

marginal & conditional:

$$\mathcal{N}(x \,|\, a, A)\ \mathcal{N}(y \,|\, b + Fx, B) = \mathcal{N}\left(\begin{array}{c|cc} x & a & A & A^\top F^\top \\ y & b + Fa\,, & FA & B + FA^\top F^\top \end{array} \right) \tag{299}$$

$$\mathcal{N}\left(\begin{array}{c|cc} x & a & A & C \\ y & b\,, & C^\top & B \end{array} \right) = \mathcal{N}(x \,|\, a, A) \cdot \mathcal{N}(y \,|\, b + C^\top A^{-1}(x\text{-}a),\ B - C^\top A^{-1}C) \tag{300}$$

$$\mathcal{N}[x \,|\, a, A]\ \mathcal{N}(y \,|\, b + Fx, B) = \mathcal{N}\left[\begin{array}{c|cc} x & a + F^\top B^{-1}b & A + F^\top B^{-1}F & -F^\top B^{-1} \\ y & B^{-1}b\ , & -B^{-1}F & B^{-1} \end{array} \right] \tag{301}$$

$$\mathcal{N}[x \,|\, a, A]\ \mathcal{N}[y \,|\, b + Fx, B] = \mathcal{N}\left[\begin{array}{c|cc} x & a + F^\top B^{-1}b & A + F^\top B^{-1}F & -F^\top \\ y & b\ , & -F & B \end{array} \right] \tag{302}$$

$$\mathcal{N}\left[\begin{array}{c|cc} x & a & A & C \\ y & b\,, & C^\top & B \end{array} \right] = \mathcal{N}[x \,|\, a - CB^{-1}b,\ A - CB^{-1}C^\top] \cdot \mathcal{N}[y \,|\, b - C^\top x, B] \tag{303}$$

$$\left| \begin{array}{cc} A & C \\ D & B \end{array} \right| = |A|\ |\widehat{B}| = |\widehat{A}|\ |B|\ \text{, where } \begin{array}{l} \widehat{A} = A - CB^{-1}D \\ \widehat{B} = B - DA^{-1}C \end{array} \tag{304}$$

$$\left[\begin{array}{cc} A & C \\ D & B \end{array} \right]^{-1} = \left[\begin{array}{cc} \widehat{A}^{-1} & -A^{-1}C\widehat{B}^{-1} \\ -\widehat{B}^{-1}DA^{-1} & \widehat{B}^{-1} \end{array} \right] \tag{305}$$

$$= \left[\begin{array}{cc} \widehat{A}^{-1} & -\widehat{A}^{-1}CB^{-1} \\ -B^{-1}D\widehat{A}^{-1} & \widehat{B}^{-1} \end{array} \right] \tag{306}$$

pair-wise belief We have a message $\alpha(x) = \mathcal{N}[x|s, S]$, transition $P(y|x) = \mathcal{N}(y|Ax + a, Q)$, and a message $\beta(y) = \mathcal{N}[y|v, V]$, what is the belief $b(y, x) = \alpha(x)P(y|x)\beta(y)$?

$$b(y, x) = \mathcal{N}[x|s, S]\ \mathcal{N}(y|Ax + a, Q^{-1})\ \mathcal{N}[y|v, V] \tag{307}$$

$$= \mathcal{N}\left[\begin{array}{c|cc} x & s & S & 0 \\ y & 0\,, & 0 & 0 \end{array} \right] \mathcal{N}\left[\begin{array}{c|cc} x & A^\top Q^{-1}a & A^\top Q^{-1}A & -A^\top Q^{-1} \\ y & Q^{-1}a\ , & -Q^{-1}A & Q^{-1} \end{array} \right] \mathcal{N}\left[\begin{array}{c|cc} x & 0 & 0 & 0 \\ y & v\,, & 0 & V \end{array} \right] \tag{308}$$

$$\propto \mathcal{N}\left[\begin{array}{c|cc} x & s + A^\top Q^{-1}a & S + A^\top Q^{-1}A & -A^\top Q^{-1} \\ y & v + Q^{-1}a\ , & -Q^{-1}A & V + Q^{-1} \end{array} \right] \tag{309}$$

Entropy

$$H(\mathcal{N}(a,A)) = \frac{1}{2}\log|2\pi e A| \tag{310}$$

Kullback-Leibler divergence

$$p = \mathcal{N}(x|a,A)\ ,\quad q = \mathcal{N}(x|b,B)\ ,\quad n = \mathsf{dim}(x)\ ,\quad D\big(p\,\|\,q\big) = \sum_x p(x)\log\frac{p(x)}{q(x)} \tag{311}$$

$$2\ D\big(p\,\|\,q\big) = \log\frac{|B|}{|A|} + \mathrm{tr}(B^{-1}A) + (b-a)^\top B^{-1}(b-a) - n \tag{312}$$

$$4\ D_{\mathsf{sym}}\big(p\,\|\,q\big) = \mathrm{tr}(B^{-1}A) + \mathrm{tr}(A^{-1}B) + (b-a)^\top(A^{-1}+B^{-1})(b-a) - 2n \tag{313}$$

λ-divergence

$$2\ D_\lambda\big(p\,\|\,q\big) = \lambda\ D\big(p\,\|\,\lambda p + (1-\lambda)q\big)\ +\ (1-\lambda)\ D\big(p\,\|\,(1-\lambda)p + \lambda q\big) \tag{314}$$

For $\lambda = .5$: Jensen-Shannon divergence.

Log-likelihoods

$$\log\mathcal{N}(x|a,A) = -\frac{1}{2}\Big[log|2\pi A| + (x\text{-}a)^\top\ A^{-1}\ (x\text{-}a)\Big] \tag{315}$$

$$\log\mathcal{N}[x|a,A] = -\frac{1}{2}\Big[log|2\pi A^{-1}| + a^\top A^{-1}a + x^\top A x - 2x^\top a\Big] \tag{316}$$

$$\sum_x \mathcal{N}(x|b,B)\log\mathcal{N}(x|a,A) = -D\big(\mathcal{N}(b,B)\,\|\,\mathcal{N}(a,A)\big) - H(\mathcal{N}(b,B)) \tag{317}$$

Mixture of Gaussians Collapsing a MoG into a single Gaussian

$$\operatorname*{argmin}_{b,B} D\big(\sum_i p_i\ \mathcal{N}(a_i,A_i)\,\|\,\mathcal{N}(b,B)\big) \tag{318}$$

$$= \quad \Big(b = \sum_i p_i a_i\ ,\ B = \sum_i p_i(A_i + a_i a_i^\top - b\,b^\top)\Big) \tag{319}$$

B Further

- Differential Geometry

  Emphasize strong relation between a Riemannian metric (and respective geodesic) and cost (in an optimization formulation). Pullbacks and costs. Only super brief, connections.

- Manifolds

  Local tangent spaces, connection. example of kinematics

- Lie groups

  exp and log

- Information Geometry

  [Integrate notes on information geometry]

References

J. M. Hernández-Lobato, M. W. Hoffman, and Z. Ghahramani. Predictive entropy search for efficient global optimization of black-box functions. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 918–926. Curran Associates, Inc., 2014. URL http://papers.nips.cc/paper/5324-predictive-entropy-search-for-efficient-global-optimization-of-blac pdf.

D. Jones, M. Schonlau, and W. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13:455–492, 1998.

M. Toussaint. A novel augmented lagrangian approach for inequalities and convergent any-time non-central updates. e-Print arXiv:1412.4329, 2014.

Index