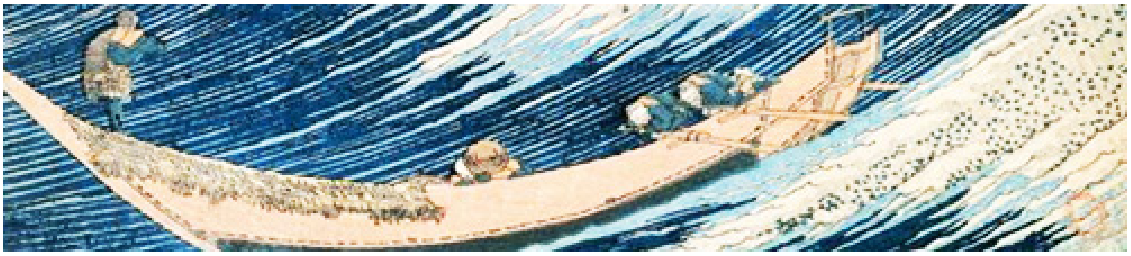
Mathematical Foundations of Data Sciences



Gabriel Peyré
CNRS & DMA
École Normale Supérieure
gabriel.peyre@ens.fr
https://mathematical-tours.github.io
www.numerical-tours.com

December 25, 2017

Chapter 17

Deep Learning

Before detailing deep architectures and their use, we start this chapter by presenting two essential computational tools that are used to train these models: stochastic optimization methods and automatic differentiation. In practice, they work hand-in-hand to be able to learn painlessly complicated non-linear models on large-scale datasets.

17.1 Stochastic Optimization

We detail some important stochastic Gradient Descent methods, which enable to perform optimization in the setting where the number of samples n is large and even infinite.

17.1.1 Minimizing Sums and Expectation

A large class of functionals in machine learning can be expressed as minimizing large sums of the form

$$\min_{\beta \in \mathbb{R}^p} \mathcal{E}(\beta) \overset{\text{def.}}{=} \frac{1}{n} \sum_{i=1}^{n} \mathcal{E}_i(\beta) \tag{17.1}$$

or even expectations of the form

$$\min_{\beta \in \mathbb{R}^p} \mathcal{E}(\beta) \overset{\text{def.}}{=} \mathbb{E}_{\mathbf{z} \sim \pi}(\mathcal{E}(\beta, \mathbf{z})) = \int_{\mathcal{Z}} \mathcal{E}(\beta, z) \mathrm{d}\pi(z). \tag{17.2}$$

Problem (17.1) can be seen as a special case of (17.2), when using a discrete empirical uniform measure $\pi = \sum_{i=1}^n \delta_i$ and setting $\mathcal{E}(x, i) = \mathcal{E}_i(x)$. One can also viewed (17.1) as a discretized "empirical" version of (17.2) when drawing $(z_i)_i$ i.i.d. according to $\mathbf{z}$ and defining $\mathcal{E}_i(x) = \mathcal{E}(x, z_i)$. In this setup, (17.1) converges to (17.2) as $n \to +\infty$.

A typical example of such a class of problems is empirical risk minimization (here without regularization $J = 0$ for simplicity) (16.7) and its expectation version (16.8), where in these cases

$$\mathcal{E}_i(\beta) = L(f(x_i, \beta), y_i) \quad \text{and} \quad \mathcal{E}_i(\beta, z) = L(f(x, \beta), y) \tag{17.3}$$

for $z = (x, y) \in \mathcal{Z} = (\mathcal{X} = \mathbb{R}^p) \times (\mathcal{Y} = \mathbb{R}^q)$ (typically $q = 1$). We illustrate bellow the methods on binary logistic classification, where

$$L(s, y) \overset{\text{def.}}{=} \log(1 + \exp(-sy)) \quad \text{and} \quad f(x, \beta) = \langle x,\, \beta \rangle, \tag{17.4}$$

see Section 16.4.2 for details. But this extends to arbitrary parametric models, and in particular deep neural networks as detailed in Section 17.3.

While some algorithms (in particular batch gradient descent) are specific to finite sums (17.1), the stochastic methods we detail next work verbatim (with the same convergence guarantees) in the expectation case (17.2). For the sake of simplicity, we however do the exposition for the finite sums case, which is sufficient in the vast majority of cases. But one should keep in mind that n can be arbitrarily large, so it is not acceptable in this setting to use algorithms whose complexity per iteration depend on n.

The general idea underlying stochastic optimization methods is *not* to have faster algorithms with respect to traditional optimization schemes such as those detailed in Chapter 13. In almost all cases, if n is not too large so that one afford the price of doing a few non-stochastic iterations, then deterministic methods are faster. But if n is so large that one cannot do even a single deterministic iteration, then stochastic methods allow one to have a fine grained scheme by breaking the cost of determinstic iterations in smaller chunks. Another advantage is that they are quite easy to parallelize.

17.1.2 Batch Gradient Descent (BGD)

The usual deterministic (batch) gradient descent (BGD) is studied in details in Section 13.1. Its iterations read

$$\beta^{(\ell+1)} = \beta^{(\ell)} - \tau_\ell \nabla \mathcal{E}(\beta^{(\ell)})$$

and the step size should be chosen as $0 < \tau_{\min} < \tau_\ell < \tau_{\max} \overset{\text{def.}}{=} 2/L$ where L is the Lipschitz constant of the gradient $\nabla \mathcal{E}$. In particular, in this determinstic setting, this step size should not go to zero and this ensures quite fast convergence (even linear rates if $\mathcal{E}$ is strongly convex).

The computation of the gradient in our setting reads

$$\nabla \mathcal{E}(\beta) = \frac{1}{n} \sum_{i=1}^{n} \nabla \mathcal{E}_i(\beta) \tag{17.5}$$

so it typically has complexity $O(np)$ if computing $\nabla \mathcal{E}_i$ has linear complexity in p.

In the ERM setting (17.3), the gradient reads

$$\nabla \mathcal{E}_i(\beta) = [\partial f(x_i, \beta)]^\top (\nabla L(f(x_i, \beta), y_i)), \tag{17.6}$$

where $\partial f(x, \beta) \in \mathbb{R}^{q \times p}$ is the Jacobian of the mapping $\beta \in \mathbb{R}^p \mapsto f(x, \beta) \in \mathbb{R}^q$, while $\nabla L(y, y') \in \mathbb{R}^q$ is the gradient with respect to the first variable, i.e. the gradient of the map $y \in \mathbb{R}^q \mapsto L(y, y') \in \mathbb{R}$.

In the case of a linear model such as (17.4), this gradient computation simply reads

$$\nabla \mathcal{E}_i(\beta) = L'(\langle x_i,\, \beta \rangle, y_i)) x_i$$

where L' is the differential of L with respect to the first variable. For the logistic loss, it is simply

$$L'(s, y) = -s \frac{e^{-sy}}{1 + e^{-sy}}.$$

17.1.3 Stochastic Gradient Descent (SGD)

For very large n, computing the full gradient $\nabla \mathcal{E}$ as in (17.5) is prohibitive. The idea of SGD is to trade this exact full gradient by an inexact proxy using a single functional $\mathcal{E}_i$ where i is drawn uniformly at random. The main idea that makes this work is that this sampling scheme provides an unbiased estimate of the gradient, in the sense that

$$\mathbb{E}_{\mathbf{i}} \nabla \mathcal{E}_{\mathbf{i}}(\beta) = \nabla \mathcal{E}(\beta) \tag{17.7}$$

where $\mathbf{i}$ is a random variable distributed uniformly in $\{1, \ldots, n\}$.

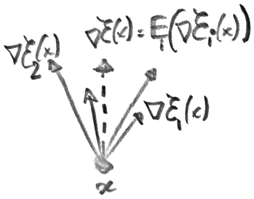
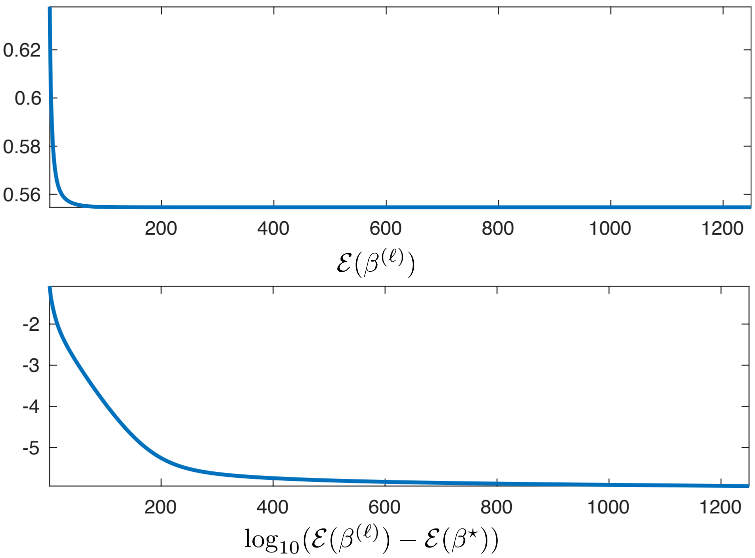

$\nabla \mathcal{E}_2(x)$
$\nabla \mathcal{E}(x) = \mathbb{E}_i(\nabla \mathcal{E}_i(x))$
$\nabla \mathcal{E}_1(x)$
x

Figure 17.2: Unbiased gradient estimate



$\mathcal{E}(\beta^{(\ell)})$

$\log_{10}(\mathcal{E}(\beta^{(\ell)}) - \mathcal{E}(\beta^\star))$

Figure 17.1: Evolution of the error of the BGD for logistic classification.

Starting from some $\beta^{(0)}$,the iterations of stochastic gradient descent (SGD) read

$$\beta^{(\ell+1)} = \beta^{(\ell)} - \tau_\ell \nabla \mathcal{E}_{i(\ell)}(\beta^{(\ell)})$$

where, for each iteration index ℓ, $i(\ell)$ is drawn uniformly at random in $\{1,\ldots,n\}$. It is important that the iterates $\beta^{(\ell+1)}$ are thus random vectors, and the theoretical analysis of the method thus studies wether this sequence of random vectors converges (in expectation or in probability for instance) toward a deterministic vector (minimizing $\mathcal{E}$), and at which speed.

Note that each step of a batch gradient descent has complexity $O(np)$, while a step of SGD only has complexity $O(p)$. SGD is thus advantageous when n is very large, and one cannot afford to do several passes through the data. In some situation, SGD can provide accurate results even with $\ell \ll n$, exploiting redundancy between the samples.

A crucial question is the choice of step size schedule τ_ℓ. It must tends to 0 in order to cancel the noise induced on the gradient by the stochastic sampling. But it should not go too fast to zero in order for the method to keep converging.

A typical schedule that ensures both properties is to have asymptotically $\tau_\ell \sim \ell^{-1}$ for $\ell \to +\infty$. We thus propose to use

$$\tau_\ell \overset{\text{def.}}{=} \frac{\tau_0}{1+\ell/\ell_0} \tag{17.8}$$

where ℓ_0 indicates roughly the number of iterations serving as a "warmup" phase.

Figure 17.3: Schematic view of SGD iterates

Figure 17.4 shows a simple 1-D example to minimize $\mathcal{E}_1(\beta)+\mathcal{E}_2(\beta)$ for $\beta \in \mathbb{R}$ and $\mathcal{E}_1(\beta) = (\beta-1)^2$ and $\mathcal{E}_2(\beta) = (\beta+1)^2$. One can see how the density of the distribution of $\beta^{(\ell)}$ progressively clusters around the minimizer $\beta^\star = 0$. Here the distribution of $\beta^{(0)}$ is uniform on $[-1/2, 1/2]$.

The following theorem shows the convergence in expectration with a $1/\sqrt{\ell}$ rate on the objective.

Theorem 52. *We assume $\mathcal{E}$ is μ-strongly convex as defined in ($\mathcal{S}_\mu$) (i.e. $\mathrm{Id}_{N\times N} \preceq \partial^2\mathcal{E}(\beta)$ if $\mathcal{E}$ is $\mathcal{C}^2$), and is such that $\|\nabla \mathcal{E}_i(x)\|^2 \leqslant C^2$. For the step size choice $\tau_\ell = \frac{1}{\mu(\ell+1)}$, one has*

$$\mathbb{E}(\|\beta^{(\ell)} - \beta^\star\|^2) \leqslant \frac{R}{\ell+1} \quad \textit{where} \quad R = \max(\|\beta^{(0)} - \beta^\star\|, C^2/\mu^2), \tag{17.9}$$

where $\mathbb{E}$ indicates an expectation with respect to the i.i.d. sampling performed at each iteration.

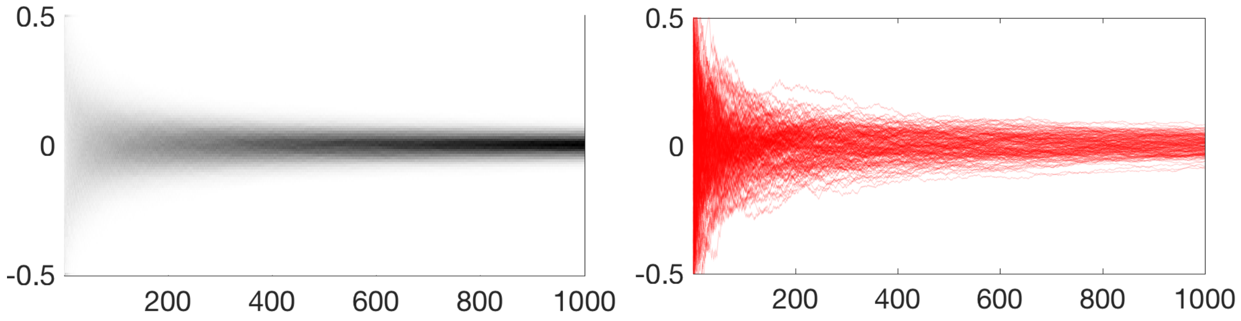


Figure 17.4: Display of a large number of trajectories $\ell \mapsto \beta^{(\ell)} \in \mathbb{R}$ generated by several runs of SGD. On the top row, each curve is a trajectory, and the bottom row displays the corresponding density.

Proof. By strong convexity, one has

$$\mathcal{E}(\beta^\star) - \mathcal{E}(\beta^{(\ell)}) \geqslant \langle \nabla\mathcal{E}(\beta^{(\ell)}),\, \beta^\star - \beta^{(\ell)} \rangle + \frac{\mu}{2}\|\beta^{(\ell)} - \beta^\star\|^2$$
$$\mathcal{E}(\beta^{(\ell)}) - \mathcal{E}(\beta^\star) \geqslant \langle \nabla\mathcal{E}(\beta^\star),\, \beta^{(\ell)} - \beta^\star \rangle + \frac{\mu}{2}\|\beta^{(\ell)} - \beta^\star\|^2.$$

Summing these two inequalities and using $\nabla\mathcal{E}(\beta^\star) = 0$ leads to

$$\langle \nabla\mathcal{E}(\beta^{(\ell)}) - \nabla\mathcal{E}(\beta^\star),\, \beta^{(\ell)} - \beta^\star \rangle = \langle \nabla\mathcal{E}(\beta^{(\ell)}),\, \beta^{(\ell)} - \beta^\star \rangle \geqslant \mu\|\beta^{(\ell)} - \beta^\star\|^2. \tag{17.10}$$

Considering only the expectation with respect to the ransom sample of $i(\ell) \sim \mathbf{i}_\ell$, one has

$$\begin{aligned}\mathbb{E}_{\mathbf{i}_\ell}(\|\beta^{(\ell+1)} - \beta^\star\|^2) &= \mathbb{E}_{\mathbf{i}_\ell}(\|\beta^{(\ell)} - \tau_\ell\nabla\mathcal{E}_{\mathbf{i}_\ell}(\beta^{(\ell)}) - \beta^\star\|^2) \\ &= \|\beta^{(\ell)} - \beta^\star\|^2 + 2\tau_\ell\langle \mathbb{E}_{\mathbf{i}_\ell}(\nabla\mathcal{E}_{\mathbf{i}_\ell}(\beta^{(\ell)})),\, \beta^\star - \beta^{(\ell)} \rangle + \tau_\ell^2\mathbb{E}_{\mathbf{i}_\ell}(\|\nabla\mathcal{E}_{\mathbf{i}_\ell}(\beta^{(\ell)})\|^2) \\ &\leqslant \|\beta^{(\ell)} - \beta^\star\|^2 + 2\tau_\ell\langle \nabla\mathcal{E}(\beta^{(\ell)})),\, \beta^\star - \beta^{(\ell)} \rangle + \tau_\ell^2 C^2\end{aligned}$$

where we used the fact (17.7) that the gradient is unbiased. Taking now the full expectation with respect to all the other previous iterates, and using (17.10) one obtains

$$\mathbb{E}(\|\beta^{(\ell+1)} - \beta^\star\|^2) \leqslant \mathbb{E}(\|\beta^{(\ell)} - \beta^\star\|^2) - 2\mu\tau_\ell\mathbb{E}(\|\beta^{(\ell)} - \beta^\star\|^2) + \tau_\ell^2 C^2 = (1 - 2\mu\tau_\ell)\mathbb{E}(\|\beta^{(\ell)} - \beta^\star\|^2) + \tau_\ell^2 C^2. \tag{17.11}$$

We show by recursion that the bound (17.9) holds. We denote $\varepsilon_\ell \overset{\text{def.}}{=} \mathbb{E}(\|\beta^{(\ell)} - \beta^\star\|^2)$. Indeed, for $\ell = 0$, this it is true that

$$\varepsilon_0 \leqslant \frac{\max(\|\beta^{(0)} - \beta^\star\|, C^2/\mu^2)}{1} = \frac{R}{1}.$$

We now assume that $\varepsilon_\ell \leqslant \frac{R}{\ell+1}$. Using (17.11) in the case of $\tau_\ell = \frac{1}{\mu(\ell+1)}$, one has, denoting $m = \ell + 1$

$$\begin{aligned}\varepsilon_{\ell+1} &\leqslant (1 - 2\mu\tau_\ell)\varepsilon_\ell + \tau_\ell^2 C^2 = \left(1 - \frac{2}{m}\right)\varepsilon_\ell + \frac{C^2}{(\mu m)^2} \\ &\leqslant \left(1 - \frac{2}{m}\right)\frac{R}{m} + \frac{R}{m^2} = \left(\frac{1}{m} - \frac{1}{m^2}\right)R = \frac{m-1}{m^2}R = \frac{m^2-1}{m^2}\frac{1}{m+1}R \leqslant \frac{R}{m+1}\end{aligned}$$

□

A weakness of SGD (as well as the SGA scheme studied next) is that it only weakly benefit from strong convexity of $\mathcal{E}$. This is in sharp contrast with BGD, which enjoy a fast linear rate for strongly convex functionals, see Theorem 35.

Figure 17.5 displays the evolution of the energy $\mathcal{E}(\beta^{(\ell)})$. It overlays on top (black dashed curve) the convergence of the batch gradient descent, with a careful scaling of the number of iteration to account for the fact that the complexity of a batch iteration is n times larger.

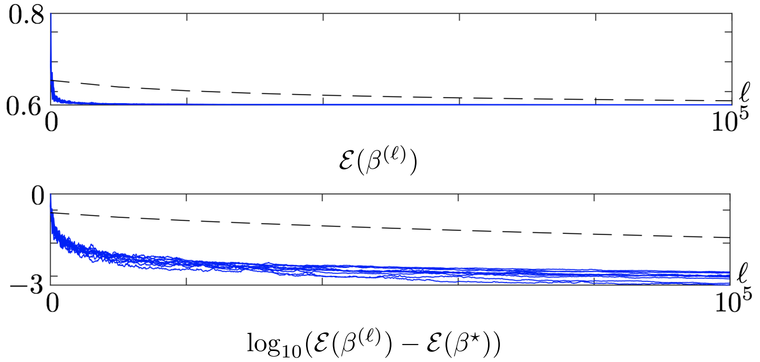


Figure 17.5: Evolution of the error of the SGD for logistic classification (dashed line shows BGD).

17.1.4 Stochastic Gradient Descent with Averaging (SGA)

Stochastic gradient descent is slow because of the fast decay of τ_ℓ toward zero. To improve somehow the convergence speed, it is possible to average the past iterate, i.e. run a "classical" SGD on auxiliary variables $(\tilde\beta^{(\ell)})_\ell$

$$\tilde\beta^{(\ell+1)} = \tilde\beta^{(\ell)} - \tau_\ell \nabla \mathcal{E}_{i(\ell)}(\tilde\beta^{(\ell)})$$

and output as estimated weight vector the Cesaro average

$$\beta^{(\ell)} \overset{\text{def.}}{=} \frac{1}{\ell} \sum_{k=1}^{\ell} \tilde\beta^{(\ell)}.$$

This defines the Stochastic Gradient Descent with Averaging (SGA) algorithm.

Note that it is possible to avoid explicitly storing all the iterates by simply updating a running average as follow

$$\beta^{(\ell+1)} = \frac{1}{\ell}\tilde\beta^{(\ell)} + \frac{\ell-1}{\ell}\beta^{(\ell)}.$$

In this case, a typical choice of decay is rather of the form

$$\tau_\ell \overset{\text{def.}}{=} \frac{\tau_0}{1+\sqrt{\ell/\ell_0}}.$$

Notice that the step size now goes much slower to 0, at rate $\ell^{-1/2}$.

Typically, because the averaging stabilizes the iterates, the choice of (ℓ_0, τ_0) is less important than for SGD.

Bach proves that for logistic classification, it leads to a faster convergence (the constant involved are smaller) than SGD, since on contrast to SGD, SGA is adaptive to the local strong convexity of E.

17.1.5 Stochastic Averaged Gradient Descent (SAG)

For problem size n where the dataset (of size $n \times p$) can fully fit into memory, it is possible to further improve the SGA method by bookkeeping the previous gradients. This gives rise to the Stochastic Averaged Gradient Descent (SAG) algorithm.

We store all the previously computed gradients in $(G^i)_{i=1}^n$, which necessitates $O(n \times p)$ memory. The iterates are defined by using a proxy g for the batch gradient, which is progressively enhanced during the iterates.

The algorithm reads

$$h \leftarrow \nabla \mathcal{E}_{i(\ell)}(\tilde\beta^{(\ell)}),$$

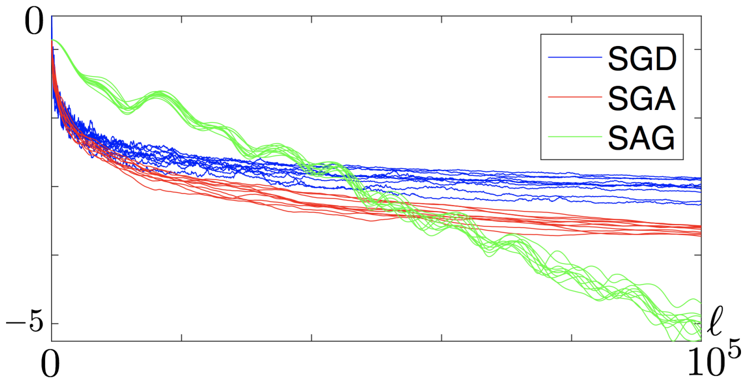


Figure 17.6: Evolution of $\log_{10}(\mathcal{E}(\beta^{(\ell)}) - \mathcal{E}(\beta^\star))$ for SGD, SGA and SAG.

$$g \leftarrow g - G^{i(\ell)} + h,$$
$$G^{i(\ell)} \leftarrow h,$$
$$\beta^{(\ell+1)} = \beta^{(\ell)} - \tau g.$$

Note that in contrast to SGD and SGA, this method uses a fixed step size τ. Similarly to the BGD, in order to ensure convergence, the step size τ should be of the order of $1/L$ where L is the Lipschitz constant of $\mathcal{E}$.

This algorithm improves over SGA and SGD since it has a convergence rate of $O(1/\ell)$ as does BGD. Furthermore, in the presence of strong convexity (for instance when X is injective for logistic classification), it has a linear convergence rate, i.e.

$$\mathbb{E}(\mathcal{E}(\beta^{(\ell)})) - \mathcal{E}(\beta^\star) = O\left(\rho^\ell\right),$$

for some $0 < \rho < 1$.

Note that this improvement over SGD and SGA is made possible only because SAG explicitly uses the fact that n is finite (while SGD and SGA can be extended to infinite n and more general minimization of expectations (17.2)).

Figure 17.9 shows a comparison of SGD, SGA and SAG.

17.2 Automatic Differentiation

The main computational bottleneck of these gradient descent methods (batch or stochastic) is the evaluation of the elementary gradients $\nabla \mathcal{E}_i$. The gradient formula (17.6) shows that it requires to remap the gradient of the loss $\nabla L(f(x_i, \beta), y_i)$ through the adjoint of the Jacobian $\partial f(x_i, \beta)$. The general idea is that for complicated model this computation should be broken in simpler sub-computation, which ultimately should corresponds to elementary operators (binary operators such as + or * and unary operators such as exp, log, etc.) for which the differential are trivial to compute.

17.2.1 Reverse Differentiation on a Feedforward Graph

To give a concrete examples which is actually found in many practical situation (and in particular for simple deep architectures, as detailed in Section 17.3.1), if the functional to be differentiated has the form

$$\mathcal{E}(\beta) = \mathcal{L} \circ \mathcal{F}_{L-1} \circ \mathcal{F}_{L-2} \circ \ldots \circ \mathcal{F}_0(\beta) \tag{17.12}$$

(often called a “feedforward” model) where $\mathcal{F}_\ell : \mathbb{R}^{n_\ell} \to \mathbb{R}^{n_{\ell+1}}$ and $\mathcal{L} : \mathbb{R}^{n_L} \to \mathbb{R}$, then one can compute the gradient $\nabla \mathcal{E}(\beta)$ in two steps:

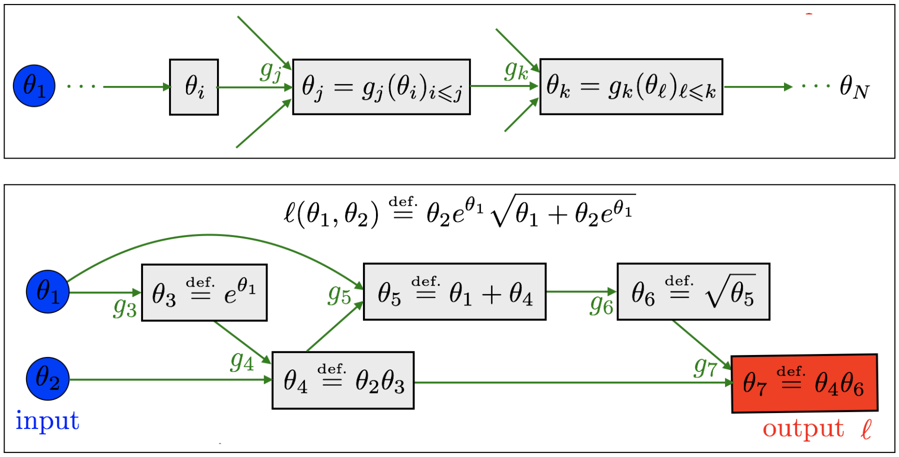

θ_1
$\cdots$
θ_i
g_j
$\theta_j = g_j(\theta_i)_{i \leqslant j}$
g_k
$\theta_k = g_k(\theta_\ell)_{\ell \leqslant k}$
$\cdots \theta_N$
$\ell(\theta_1, \theta_2) \overset{\text{def.}}{=} \theta_2 e^{\theta_1} \sqrt{\theta_1 + \theta_2 e^{\theta_1}}$
θ_1
g_3
$\theta_3 \overset{\text{def.}}{=} e^{\theta_1}$
g_5
$\theta_5 \overset{\text{def.}}{=} \theta_1 + \theta_4$
g_6
$\theta_6 \overset{\text{def.}}{=} \sqrt{\theta_5}$
θ_2
g_4
$\theta_4 \overset{\text{def.}}{=} \theta_2 \theta_3$
g_7
$\theta_7 \overset{\text{def.}}{=} \theta_4 \theta_6$
input
output ℓ

Figure 17.7: Top: elementary component of the DAG computational graph. Bottom: example of DAG computational graph.

– A forward pass, where one evaluates the function itself and keep track of all the intermediate computations, i.e., initializing $\beta_0 = \beta$, one computes

$$\beta_{\ell+1} \overset{\text{def.}}{=} \mathcal{F}_\ell(\beta_\ell) \quad \text{and} \quad \mathcal{E}(\beta) = \mathcal{L}(\beta_L).$$

– A backward pass, where one use the chain rule together with the Jacobian transposition

$$\nabla \mathcal{E}(\beta) = [\partial \mathcal{F}_0(\beta_0)]^* \circ \ldots \circ [\partial \mathcal{F}_{L-1}(\beta_{L-1})]^* \left(\nabla \mathcal{L}(\beta_L)\right) \tag{17.13}$$

to define the following backward recursion, initialized by $h_L = \nabla \mathcal{L}(\beta_L)$, and then

$$h_{\ell-1} \overset{\text{def.}}{=} [\partial \mathcal{F}_{\ell-1}(\beta_{\ell-1})]^*(h_\ell) \quad \text{and} \quad \nabla \mathcal{E}(\beta) = h_0. \tag{17.14}$$

The main issue here is when these adjoint Jacobian $[\partial \mathcal{F}_{\ell-1}(\beta_{\ell-1})]^*$ are difficult to apply (and it is out of question in most cases to actually store them on a computer, since it would occurs an enormous storage requirement and typical quadratic time scaling for the algorithms). We now detail a finer grained analysis which enable to tackle building blocks of arbitrary complexity.

The computation (17.13) should be compared with the "forward" accumulation

$$\nabla \mathcal{E}(\beta) = \partial \mathcal{L}(\beta_L) \circ \partial \mathcal{F}_{L-1}(\beta_{L-1}) \ldots \circ \partial \mathcal{F}_0(\beta_0).$$

Computing these matrix product would be extremely costly, although it would require no memory overhead because the computation would be caried over in parallel to the evaluation of the function.

17.2.2 Reverse Differentiation on a Generic Computational Graph

One can generalize the idea above to differentiate automatically any function which can be implemented on a computer. What is even more surprising is that the computational cost is the same as the one of evaluating the function itself. This fundamental computational fact (that gradient evaluation and function evaluation have the same computational cost) is not so well known, but is of paramount practical interest when it comes to differentiating complicated recursive functions. We will apply it in a very simple setup for deep-architectures, but it can be applied to much more involved computational architectures.

Note that this results only applies for function which output scalar values. For functions which output vector values, one can of course re-use this idea for each output, but this is in general vastly sub-optimal, because it ignore the redundancy between the computation of each output. The determination of optimal strategy in this case is known to be NP-hard. This include for instance the computation of the Hessian of a scalar valued function (since it corresponds to the differentiation of the gradient, which is itself a vector-valued function). Fortunately, for machine learning application, one is often interested in differentiating only empirical losses functions, which as scalar valued.

Forward pass as a DAG traversal. The crux of this idea is that the computational flow of any computable function ℓ can be represented as a directed acyclic graph. We denote $(\theta_i)_{i=1}^R$ the set of all scalar variables (input, output and intermediary) manipulated by the computational program. Without loss of generality, we impose that the first variable $(\theta_1, \ldots, \theta_M)$ are the M input variables, while the last θ_R is the output variable. The function to be computed is thus of the form

$$\theta_R = \ell(\theta_1, \ldots, \theta_M)$$

where $\ell : \mathbb{R}^M \to \mathbb{R}$ is broken in $R - M + 1$ intermediate steps corresponding to all the remaining variable $(\theta_i)_{M<i<R}$. The successive execution of the program defines an ordering of all the intermediate variables, so that, after initializing the input variables $(\theta_1, \ldots, \theta_M)$, the forward pass computes the value of θ_r for $r = M + 1, \ldots, R$ as

$$\theta_r = g_r(\theta_{\pi(r)})$$

for some scalar valued function $g_r : \mathbb{R}^{|\pi(r)|} \to \mathbb{R}$, where $\pi(r) \subset \{1, \ldots, r-1\}$ is the set of "parent" node of r in a directed acyclic graph (DAG). Figure 17.7 shows an example of such a computational DAG.

From a symbolic computation point of view, variables θ_j (for $j > M$) in the graph can be interpreted either as variables (i.e. which can be assigned scalar values) and functions depending on input variables θ_m for $m \leqslant M$. The beauty of this DAG representation is that one can also view θ_j as depending on any other intermediate variable θ_i as long as $i < j$.

Direct mode auto-diff. The goal is to compute the gradient vector, which reads

$$\nabla\ell = \left(\frac{\partial\theta_R}{\partial\theta_m}\right)_{m=1}^M.$$

The naive way to comput this gradient vector would thus be to compute for each of the M input variable θ_m the differential $\frac{\partial\theta_j}{\partial\theta_m}$ of all function θ_j with respect to θ_m. Without loss of generality, we consider $m = 1$. This can be achieved by using the standard chain rule

$$\frac{\partial\theta_j}{\partial\theta_1} = \sum_{i\in\pi(j)} \frac{\partial\theta_j}{\partial\theta_i}\frac{\partial\theta_i}{\partial\theta_1}. \tag{17.15}$$

Here the multipliers involved are actually differential of the elementary functions

$$\frac{\partial\theta_j}{\partial\theta_i} = \partial_i g_j$$

Note that this writing is abusive, since $\frac{\partial\theta_j}{\partial\theta_i}$ really means that in practice such a differential is evaluated assuming all the variable $(\theta_r)_{r<j}$ on which the function θ_j depends are defined to their respective value (which have been computed by the forward pass, which here can be run in parallel to the forward DAG traversal).

By traversing forward the DAG, iterating this formula compute all the derivative, and in particular $(\nabla\ell)_1 = \frac{\partial\theta_R}{\partial\theta_1}$. This approach, while being the most natural, is however vastly sub-optimal because its complexity is M times the one of the evaluation of the function.

forward

```
function $\ell(\theta_1, \ldots, \theta_M)$
    for $r = M+1, \ldots, R$
        $\theta_r = g_r(\theta_{\text{Parents}(r)})$
    return $\theta_R$
```

backward

```
function $\nabla\ell(\theta_1, \ldots, \theta_M)$
    $\nabla_R \ell = 1$
    for $r = R-1, \ldots, 1$
        $\nabla_r \ell = \sum_{s \in \text{Child}(r)} \partial_r g_s(\theta)\, \nabla_s \ell$
    return $(\nabla_1 \ell, \ldots, \nabla_M \ell)$
```

Figure 17.8: Recap of the two step of the automatic differentiation procedure.

Reverse mode auto-diff. Instead of computing the quantities $(\frac{\partial\theta_j}{\partial\theta_1})_j$, a radically different approach consists in rather computing the quantities $\frac{\partial\theta_L}{\partial\theta_j} = (\nabla\ell)_j$. In place of the "forward" chain rule, one needs to the backward one

$$\frac{\partial\theta_R}{\partial\theta_j} = \sum_{j\in\pi(k)} \frac{\partial\theta_R}{\partial\theta_k}\frac{\partial\theta_k}{\partial\theta_j}. \tag{17.16}$$

Note that here the summation is done over k which are "child" of j in the DAG. Here the multiplier appearing in the formula are differential of the elementary function since $\frac{\partial\theta_k}{\partial\theta_j} = \partial_j g_k$. The main interest of this reverse recursion (17.16) with respect to the direct one (17.16) is that it only needs to be run once, so that the overall complexity is the same as the one of the forward pass to compute the function itself.

Figure 17.8 recaps the two passes of the reverse mode automatic differentiation method.

The main bottleneck of this backward automatic differentiation technic is the memory consumption. Indeed, since all intermediate results need to be computed and stored explicitly before applying the backward pass, memory grows proportionally to execution time. This can be unacceptable for very large machine learning model. Fortunately, it is possible to trade time vs. memory and only keep track of a fraction of intermediate results, and retrieve the missing result locally by small forward passes. Doing this approach recursively allows to only have a logarithmic overhead in term of both time and memory, showing the vast superiority of automatic differentiation method with respect to any other alternative for differentiation. We could not insist more on the crucial importance and impact of this class of technics on modern data science.

17.3 Deep Discriminative Models

17.3.1 Deep Network Structure

Deep learning are estimator $f(x, \beta)$ which are built as composition of simple building blocks. In their simplest form (non-recursive), they corresponds to a simple linear computational graph as already defined in (17.12) (without the loss $\mathcal{L}$), and we write this as

$$f(\cdot, \beta) = f_{L-1}(\cdot, \beta_1) \circ f_{L-2}(\cdot, \beta_2) \circ \ldots \circ f_0(\cdot, \beta_0)$$

where $\beta = (\beta_0, \ldots, \beta_{L-1})$ is the set of parameters, and

$$f_\ell(\cdot, \beta_\ell) : \mathbb{R}^{n_\ell} \to \mathbb{R}^{n_{\ell+1}}$$

While it is possible to consider more complicated architecture (in particular recurrent ones), we restrict here out attention to these simple linear graph computation structures (so-called feedforward networks).

The supervised learning of these parameters β is usually done by empirical risk minimization (16.7) using SGD-type methods as explained in Section 17.1. Note that this results in highly non-convex optimization problems. In particular, strong convergence guarantees such as Theorem 52 do not hold anymore, and only weak convergence (toward stationary points) holds. SGD type technics are however found to work

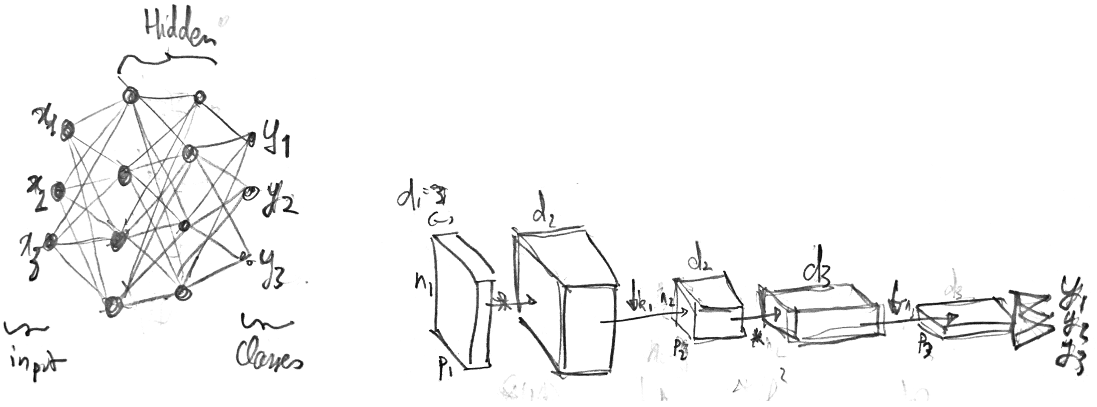


Figure 17.9: Left: example of fully connected network. Right: example of convolutional neural network.

surprisingly well in practice, and it now believe that the success of these deep-architecture approaches (in particular the ability of these over-parameterized model to generalize well) are in large part due to the dynamics of the SGD itself, which induce an implicit regularization effect.

For these simple linear architectures, the gradient of the ERM loss (17.6) can be computed using the reverse mode computation detailed in Section 17.2.1. In particular, in the context of deep learning, formula (17.18). One should however keep in mind that for more complicated (e.g. recursive) architectures, such a simple formula is not anymore available, and one should resort to reverse mode automatic differentiation (see Section 17.2.2), which, while being conceptually simple, is actually implementing possibly highly non-trivial and computationally optimal recursive differentiation.

In most successful applications of deep-learning, each computational block $f_\ell(\cdot, \beta_\ell)$ is actually very simple, and is the composition of

- an affine map, $B_\ell \cdot + b_\ell$ with a matrix $B_\ell \in \mathbb{R}^{n_\ell \times \tilde{n}_\ell}$ and a vector $b_\ell \in \mathbb{R}^{\tilde{n}_\ell}$ parametrized (in most case linearly) by β_ℓ,
- a fixed (not depending on β_ℓ) non-linearity $\rho_\ell : \mathbb{R}^{\tilde{n}_\ell} \to \mathbb{R}^{n_{\ell+1}}$

which we write as

$$\forall\, x_\ell \in \mathbb{R}^{n_\ell}, \quad f_\ell(x_\ell, \beta_\ell) = \rho_\ell(B_\ell x_\ell + b_\ell) \in \mathbb{R}^{n_{\ell+1}}. \tag{17.17}$$

In the simplest case, the so-called "fully connected", one has $(B_\ell, b_\ell) = \beta_\ell$, i.e. B_ℓ is a full matrix and its entries (together with the bias b_ℓ) are equal to the set of parameters β_ℓ. Also in the simplest cases ρ_ℓ is a pointwise non-linearity $\rho_\ell(z) = (\tilde{\rho}_\ell(z_k))_k$, where $\tilde{\rho}_\ell : \mathbb{R} \to \mathbb{R}$ is non-linear. The most usual choices are the rectified linear unit (ReLu) $\tilde{\rho}_\ell(s) = \max(s, 0)$ and the sigmoid $\tilde{\rho}_\ell(s) = \theta(s) = (1 + e^{-s})^{-1}$.

The important point here is that the interleaving of non-linear map progressively increases the complexity of the function $f(\cdot, \beta)$.

The parameter $\beta = (B_\ell, b_\ell)_\ell$ of such a deep network are then trained by minimizing the ERM functional (16.7) using SGD-type stochastic optimization method. The gradient can be computed efficiently (with complexity proportional to the application of the model, i.e. $O(\sum_\ell n_\ell^2)$) by automatic differentiation. Since such models are purely feedforward, one can directly use the back-propagation formula (17.12).

For regression tasks, one can can directly use the output of the last layer (using e.g. a ReLu non-linearity) in conjunction with a ℓ^2 squared loss L. For classification tasks, the output of the last layer needs to be transformed into class probabilities by a multi-class logistic map (**??**).

An issue with such a fully connected setting is that the number of parameters is too large to be applicable to large scale data such as images. Furthermore, it ignores any prior knowledge about the data, such as for instance some invariance. This is addressed in more structured architectures, such as for instance convolutional networks detailed in Section 17.3.3.

17.3.2 Perceptron and Shallow Models

Before going on with the description of deep architectures, let us re-interpret the logistic classification method detailed in Sections 16.4.2 and 16.4.3.

The two-class logistic classification model (16.16) is equal to a single layer ($L = 1$) network of the form (17.17) (ignoring the constant bias term) where

$$B_0 x = \langle x,\, \beta\rangle \quad \text{and} \quad \tilde{\lambda}_0(u) = \theta(u).$$

The resulting one-layer network $f(x, \beta) = \theta(\langle x,\, \beta\rangle)$ (possibly including a bias term by adding one dummy dimension to x) is trained using the loss, for binary classes $y \in \{0, 1\}$

$$L(t, y) = -\log(t^y(1-t)^{1-y}) = -y\log(t) - (1-y)\log(1-t).$$

In this case, the ERM optimization is of course a convex program.

Multi-class models with K classes are obtained by computing $B_0 x = (\langle x,\, \beta_k\rangle)_{k=1}^K$, and a normalized logistic map

$$f(x, \beta) = \mathcal{N}((\exp(\langle x,\, \beta_k\rangle))_k) \quad \text{where} \quad \mathcal{N}(u) = \frac{u}{\sum_k u_k}$$

and assuming the classes are represented using vectors y on the probability simplex, one should use as loss

$$L(t, y) = -\sum_{k=1}^K y_k \log(t_k).$$

17.3.3 Convolutional Neural Networks

In order to be able to tackle data of large size, and also to improve the performances, it is important to leverage some prior knowledge about the structure of the typical data to process. For instance, for signal, images or videos, it is important to make use of the spacial location of the pixels and the translation invariance (up to boundary handling issues) of the domain.

Convolutional neural networks are obtained by considering that the manipulated vectors $x_\ell \in \mathbb{R}^{n_\ell}$ at depth ℓ in the network are of the form $x_\ell \in \mathbb{R}^{\bar{n}_\ell \times d_\ell}$, where $\bar{n}_\ell$ is the number of "spatial" positions (typically along a 1-D, 2-D, or 3-D grid) and d_ℓ is the number of "channels". For instance, for color images, one starts with $\tilde{n}_\ell$ being the number of pixels, and $d_\ell = 3$.

The linear operator $B_\ell : \mathbb{R}^{\bar{n}_\ell \times d_\ell} \to \mathbb{R}^{\bar{n}_\ell \times d_{\ell+1}}$ is then (up to boundary artefact) translation invariant and hence a convolution along each channel (note that the number of channels can change between layers). It is thus parameterized by a set of filters $(\psi_{\ell,r,s})_{s=1,\ldots,d_\ell}^{r=1,\ldots,d_{\ell+1}}$. Denoting $x_\ell = (x_{\ell,s,\cdot})_{s=1}^{d_\ell}$ the different layers composing x_ℓ, the linear map reads

$$\forall\, r \in \{1, \ldots, d_{\ell+1}\}, \quad (B_\ell x_\ell)_{r,\cdot} = \sum_{s=1}^{d_\ell} \psi_{\ell,r,s} \star x_{\ell,s,\cdot}$$

and the bias term $b_\ell \in \mathbb{R}$ is contant (to maintain translation invariance).

The non-linear maps across layers serve two purposes: as before a pointwise non-linearity is applied, and then a sub-sampling helps to reduce the computational complexity of the network. This is very similar to the construction of the fast wavelet transform. Denoting by m_k the amount of down-sampling, where usually $m_k = 1$ (no reduction) or $m_k = 2$ (reduction by a factor two in each direction). One has

$$\lambda_\ell(u) = \left(\tilde{\lambda}_\ell(u_{s, m_k \cdot})\right)_{s=1\ldots,d_{\ell+1}}.$$

In the literature, it has been proposed to replace linear sub-sampling by non-linear sub-sampling, for instance the so-called max-pooling (that operate by taking the maximum among groups of m_ℓ successive values), but it seems that linear sub-sampling is sufficient in practice when used in conjunction with very deep (large L) architectures.

The intuition behind such model is that as one moves deeper through the layers, the neurons are receptive to larger areas in the image domain (although, since the transform is non-linear, precisely giving sense to this statement and defining a proper "receptive field" is non-trivial). Using an increasing number of channels helps to define different classes of "detectors" (for the first layer, they detect simple patterns such as edges and corner, and progressively capture more elaborated shapes).

In practice, the last few layers (2 or 3) of such a CNN architectures are chosen to be fully connected. This is possible because, thanks to the sub-sampling, the dimension of these layers are small.

The parameters of such a model are the filters $\beta = (\psi_{\ell,r,s})_{\ell,s,r}$, and they are trained by minimizing the ERM functional (16.7). The gradient is typically computed by backpropagation. Indeed, when computing the gradient with respect to some filter $\psi_{\ell,r,s}$, the feedforward computational graph has the form (17.12). For simplicity, we re-formulate this computation in the case of a single channel per layer (multiple layer can be understood as replacing convolution by matrix-domain convolution). The forward pass computes all the inner coefficients, by traversing the network from $\ell = 0$ to $\ell = L - 1$,

$$x_{\ell+1} = \lambda_\ell(\psi_\ell \star x_\ell)$$

where $\lambda_\ell(u) = (\tilde{\lambda}_\ell(u_i))_i$ is applied component wise. Then, denoting $\mathcal{E}(\beta) = \mathcal{L}(\beta, y)$ the loss to be minimized with respect to the set of filters $\beta = (\psi_\ell)_\ell$, and denoting $\nabla_\ell \mathcal{E}(\beta) = \frac{\partial \mathcal{E}(\beta)}{\partial \psi_\ell}$ the gradient with respect to ψ_ℓ, one computes all the gradients by traversing the network in reverse order, from $\ell = L - 1$ to $\ell = 0$

$$\nabla_\ell \mathcal{E}(\beta) = [\lambda'_\ell(\psi_\ell \star x_\ell)] \odot [\bar{\psi}_\ell \star \nabla_{\ell+1}\mathcal{E}(\beta)], \tag{17.18}$$

where $\lambda'_\ell(u) = (\tilde{\lambda}'_\ell(u_i))_i$ applies the derivative of $\tilde{\lambda}_\ell$ component wise, and where $\bar{\psi}_\ell = \psi_\ell(-\cdot)$ is the reversed filter. Here, $\odot$ is the pointwise multiplication of vectors. The recursion is initialized as $\nabla \mathcal{E}_L(\beta) = \nabla \mathcal{L}(x_L, y)$, the gradient of the loss itself.

This recursion (17.18) is the celebrated backpropagation algorithm put forward by Yann Lecun. Note that to understand and code these iterations, one does not need to rely on the advanced machinery of reverse mode automatic differentiation exposed in Section 17.2.2. The general automatic differentiation method is however crucial to master because advanced deep-learning architectures are not purely feedforward, and might include recursive connexions. Furthermore, automatic differentiation is useful outside deep learning, and considerably eases prototyping for modern data-sciences with complicated non-linear models.

17.3.4 Scattering Transform

The scattering transform, introduced by Mallat and his collaborators, is a specific instance of deep convolutional network, where the filters $(\psi_{\ell,r,s})_{\ell,s,r}$ are not trained, and are fixed to be wavelet filters. This network can be understood as a non-linear extension of the wavelet transform. In practice, the fact that it is fixed prevent it to be applied to arbitrary data (and is used mostly on signals and images) and it does not lead to state of the art results for natural images. Nevertheless, it allows to derives some regularity properties about the feature extraction map $f(\cdot, \beta)$ computed by the network in term of stability to diffeomorphisms. It can also be used as a set of fixed initial features which can be further enhanced by a trained deep network, as shown by Edouard Oyallon.

Bibliography

[1] P. Alliez and C. Gotsman. Recent advances in compression of 3d meshes. In N. A. Dodgson, M. S. Floater, and M. A. Sabin, editors, *Advances in multiresolution for geometric modelling*, pages 3–26. Springer Verlag, 2005.

[2] P. Alliez, G. Ucelli, C. Gotsman, and M. Attene. Recent advances in remeshing of surfaces. In *AIM@SHAPE repport*. 2005.

[3] Amir Beck. *Introduction to Nonlinear Optimization: Theory, Algorithms, and Applications with MATLAB*. SIAM, 2014.

[4] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.

[5] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[6] E. Candès and D. Donoho. New tight frames of curvelets and optimal representations of objects with piecewise C^2 singularities. *Commun. on Pure and Appl. Math.*, 57(2):219–266, 2004.

[7] E. J. Candès, L. Demanet, D. L. Donoho, and L. Ying. Fast discrete curvelet transforms. *SIAM Multiscale Modeling and Simulation*, 5:861–899, 2005.

[8] A. Chambolle. An algorithm for total variation minimization and applications. *J. Math. Imaging Vis.*, 20:89–97, 2004.

[9] Antonin Chambolle, Vicent Caselles, Daniel Cremers, Matteo Novaga, and Thomas Pock. An introduction to total variation for image analysis. *Theoretical foundations and numerical methods for sparse recovery*, 9(263-340):227, 2010.

[10] Antonin Chambolle and Thomas Pock. An introduction to continuous optimization for imaging. *Acta Numerica*, 25:161–319, 2016.

[11] S.S. Chen, D.L. Donoho, and M.A. Saunders. Atomic decomposition by basis pursuit. *SIAM Journal on Scientific Computing*, 20(1):33–61, 1999.

[12] F. R. K. Chung. Spectral graph theory. *Regional Conference Series in Mathematics, American Mathematical Society*, 92:1–212, 1997.

[13] Philippe G Ciarlet. Introduction à l'analyse numérique matricielle et à l'optimisation. 1982.

[14] P. L. Combettes and V. R. Wajs. Signal recovery by proximal forward-backward splitting. *SIAM Multiscale Modeling and Simulation*, 4(4), 2005.

[15] P. Schroeder et al. D. Zorin. Subdivision surfaces in character animation. In *Course notes at SIGGRAPH 2000*, July 2000.

[16] I. Daubechies, M. Defrise, and C. De Mol. An iterative thresholding algorithm for linear inverse problems with a sparsity constraint. *Commun. on Pure and Appl. Math.*, 57:1413–1541, 2004.

[17] I. Daubechies and W. Sweldens. Factoring wavelet transforms into lifting steps. *J. Fourier Anal. Appl.*, 4(3):245–267, 1998.

[18] D. Donoho and I. Johnstone. Ideal spatial adaptation via wavelet shrinkage. *Biometrika*, 81:425–455, Dec 1994.

[19] Heinz Werner Engl, Martin Hanke, and Andreas Neubauer. *Regularization of inverse problems*, volume 375. Springer Science & Business Media, 1996.

[20] M. Figueiredo and R. Nowak. An EM Algorithm for Wavelet-Based Image Restoration. *IEEE Trans. Image Proc.*, 12(8):906–916, 2003.

[21] M. S. Floater and K. Hormann. Surface parameterization: a tutorial and survey. In N. A. Dodgson, M. S. Floater, and M. A. Sabin, editors, *Advances in multiresolution for geometric modelling*, pages 157–186. Springer Verlag, 2005.

[22] Simon Foucart and Holger Rauhut. *A mathematical introduction to compressive sensing*, volume 1. Birkhäuser Basel, 2013.

[23] I. Guskov, W. Sweldens, and P. Schröder. Multiresolution signal processing for meshes. In Alyn Rockwood, editor, *Proceedings of the Conference on Computer Graphics (Siggraph99)*, pages 325–334. ACM Press, August8–13 1999.

[24] A. Khodakovsky, P. Schröder, and W. Sweldens. Progressive geometry compression. In *Proceedings of the Computer Graphics Conference 2000 (SIGGRAPH-00)*, pages 271–278, New York, July 23–28 2000. ACMPress.

[25] L. Kobbelt. $\sqrt{3}$ subdivision. In Sheila Hoffmeyer, editor, *Proc. of SIGGRAPH'00*, pages 103–112, New York, July 23–28 2000. ACMPress.

[26] M. Lounsbery, T. D. DeRose, and J. Warren. Multiresolution analysis for surfaces of arbitrary topological type. *ACM Trans. Graph.*, 16(1):34–73, 1997.

[27] S. Mallat. *A Wavelet Tour of Signal Processing, 3rd edition.* Academic Press, San Diego, 2009.

[28] Stephane Mallat. *A wavelet tour of signal processing: the sparse way.* Academic press, 2008.

[29] D. Mumford and J. Shah. Optimal approximation by piecewise smooth functions and associated variational problems. *Commun. on Pure and Appl. Math.*, 42:577–685, 1989.

[30] Neal Parikh, Stephen Boyd, et al. Proximal algorithms. *Foundations and Trends® in Optimization*, 1(3):127–239, 2014.

[31] Gabriel Peyré. *L'algèbre discrète de la transformée de Fourier*. Ellipses, 2004.

[32] J. Portilla, V. Strela, M.J. Wainwright, and Simoncelli E.P. Image denoising using scale mixtures of Gaussians in the wavelet domain. *IEEE Trans. Image Proc.*, 12(11):1338–1351, November 2003.

[33] E. Praun and H. Hoppe. Spherical parametrization and remeshing. *ACM Transactions on Graphics*, 22(3):340–349, July 2003.

[34] L. I. Rudin, S. Osher, and E. Fatemi. Nonlinear total variation based noise removal algorithms. *Phys. D*, 60(1-4):259–268, 1992.

[35] Otmar Scherzer, Markus Grasmair, Harald Grossauer, Markus Haltmeier, Frank Lenzen, and L Sirovich. *Variational methods in imaging.* Springer, 2009.

[36] P. Schröder and W. Sweldens. Spherical Wavelets: Efficiently Representing Functions on the Sphere. In *Proc. of SIGGRAPH 95*, pages 161–172, 1995.

[37] P. Schröder and W. Sweldens. Spherical wavelets: Texture processing. In P. Hanrahan and W. Purgathofer, editors, *Rendering Techniques '95*. Springer Verlag, Wien, New York, August 1995.

[38] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[39] A. Sheffer, E. Praun, and K. Rose. Mesh parameterization methods and their applications. *Found. Trends. Comput. Graph. Vis.*, 2(2):105–171, 2006.

[40] Jean-Luc Starck, Fionn Murtagh, and Jalal Fadili. *Sparse image and signal processing: Wavelets and related geometric multiscale analysis.* Cambridge university press, 2015.

[41] W. Sweldens. The lifting scheme: A custom-design construction of biorthogonal wavelets. *Applied and Computation Harmonic Analysis*, 3(2):186–200, 1996.

[42] W. Sweldens. The lifting scheme: A construction of second generation wavelets. *SIAM J. Math. Anal.*, 29(2):511–546, 1997.