Course notes on Automatic Differentiation

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

May 18, 2020

Abstract

These are course note on automatic differentiation. They cover the basics of the forward and backward modes, and draw connexions with the adjoint state method.

The main computational bottleneck of gradient descent methods (batch or stochastic) is the computation of gradients $\nabla f(x)$. For simple functionals, such as those encountered in ERM for linear models, and also for MLP with a single hidden layer, it is possible to compute these gradients in closed form, and that the main computational burden is the evaluation of matrix-vector products. For more complicated functionals (such as those involving deep networks), computing the formula for the gradient quickly becomes cumbersome. Even worse: computing these gradients using the usual chain rule formula is sub-optimal. We presents methods to compute recursively in an optimal manner these gradients. The purpose of this approach is to automatize this computational step.

1 Finite Differences and Symbolic Calculus

We consider $f : \mathbb{R}^p \to \mathbb{R}$ and want to derive a method to evaluate $\nabla f : \mathbb{R}^p \mapsto \mathbb{R}^p$. Approximating this vector field using finite differences, i.e. introducing $\varepsilon > 0$ small enough and computing

$$\frac{1}{\varepsilon}(f(x+\varepsilon\delta_1) - f(x), \ldots, f(x+\varepsilon\delta_p) - f(x))^\top \approx \nabla f(x)$$

requires $p+1$ evaluations of f, where we denoted $\delta_k = (0, \ldots, 0, 1, 0, \ldots, 0)$ where the 1 is at index k. For a large p, this is prohibitive. The method we describe in this section (the so-called reverse mode automatic differentiation) has in most cases a cost proportional to a single evaluation of f. This type of method is similar to symbolic calculus in the sense that it provides (up to machine precision) exact gradient computation. But symbolic calculus does not takes into account the underlying algorithm which compute the function, while automatic differentiation factorizes the computation of the derivative according to an efficient algorithm.

2 Computational Graphs

We consider a generic function $f(x)$ where $x = (x_1, \ldots, x_s)$ are the input variables. We assume that f is implemented in an algorithm, with intermediate variable $(x_{s+1}, \ldots, x_t)$ where t is the total number of variables. The output is x_t, and we thus denote $x_t = f(x)$ this function. We denote $x_k \in \mathbb{R}^{n_k}$ the

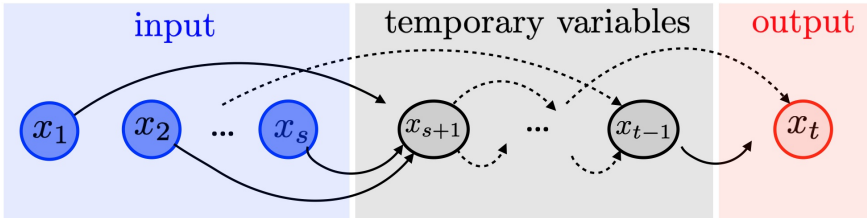

Figure 1: A computational graph.

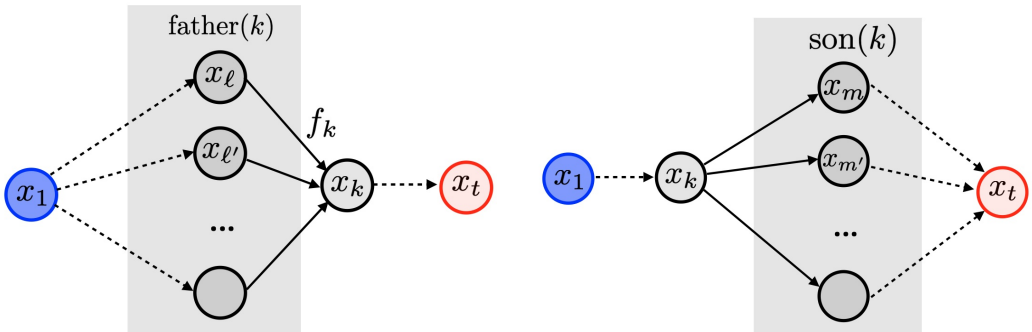

Figure 2: Relation between the variable for the forward (left) and backward (right) modes.

dimensionality of the variables. The goal is to compute the derivatives $\frac{\partial f(x)}{\partial x_k} \in \mathbb{R}^{n_t \times n_k}$ for $k = 1, \dots, s$. For the sake of simplicity, one can assume in what follows that $n_k = 1$ so that all the involved quantities are scalar (but if this is not the case, beware that the order of multiplication of the matrices of course matters).

A numerical algorithm can be represented as a succession of functions of the form

$$\forall\, k = s+1, \dots, t, \quad x_k = f_k(x_1, \dots, x_{k-1})$$

where f_k is a function which only depends on the previous variables, see Fig. 1. One can represent this algorithm using a directed acyclic graph (DAG), linking the variables involved in f_k to x_k. The node of this graph are thus conveniently ordered by their indexing, and the directed edges only link a variable to another one with a strictly larger index. The evaluation of $f(x)$ thus corresponds to a forward traversal of this graph. Note that the goal of automatic differentiation is not to define an efficient computational graph, it is up to the user to provide this graph. Computing an efficient graph associated to a mathematical formula is a complicated combinatorial problem, which still has to be solved by the user. Automatic differentiation thus leverage the availability of an efficient graph to provide an efficient algorithm to evaluate derivatives.

3 Forward Mode of Automatic Differentiation

The forward mode correspond to the usual way of computing differentials. It compute the derivative $\frac{\partial x_k}{\partial x_1}$ of all variables x_k with respect to x_1. One then needs to repeat this method p times to compute all the derivative with respect to $x_1, x_2, \dots, x_p$ (we only write thing for the first variable, the method being of course the same with respect to the other ones).

The method initialize the derivative of the input nodes

$$\frac{\partial x_1}{\partial x_1} = \mathrm{Id}_{n_1 \times n_1}, \quad \frac{\partial x_2}{\partial x_1} = 0_{n_2 \times n_1}, \dots, \quad \frac{\partial x_s}{\partial x_1} = 0_{n_s \times n_1},$$

(and thus 1 and 0's for scalar variables), and then iteratively make use of the following recursion formula

$$\forall\, k = s+1,\ldots,t, \quad \frac{\partial x_k}{\partial x_1} = \sum_{\ell \in \mathrm{parent}(k)} \left[\frac{\partial x_k}{\partial x_\ell}\right] \times \frac{\partial x_\ell}{\partial x_1} = \sum_{\ell \in \mathrm{parent}(k)} \frac{\partial f_k}{\partial x_\ell}(x_1,\ldots,x_{k-1}) \times \frac{\partial x_\ell}{\partial x_1}.$$

The notation "parent(k)" denotes the nodes $\ell < k$ of the graph that are connected to k, see Figure 2, left. Here the quantities being computed (i.e. stored in computer variables) are the derivatives $\frac{\partial x_\ell}{\partial x_1}$, and $\times$ denotes in full generality matrix-matrix multiplications. We have put in $[\ldots]$ an informal notation, since here $\frac{\partial x_k}{\partial x_\ell}$ should be interpreted not as a numerical variable but needs to be interpreted as derivative of the function f_k, which can be evaluated on the fly (we assume that the derivative of the function involved are accessible in closed form).

Assuming all the involved functions $\frac{\partial f_k}{\partial x_k}$ have the same complexity (which is likely to be the case if all the n_k are for instance scalar or have the same dimension), and that the number of parent node is bounded, one sees that the complexity of this scheme is p times the complexity of the evaluation of f (since this needs to be repeated p times for $\frac{\partial}{\partial x_1},\ldots,\frac{\partial}{\partial x_p}$). For a large p, this is prohibitive.

Simple example. We consider the fonction

$$f(x,y) = y\log(x) + \sqrt{y\log(x)} \tag{1}$$

whose computational graph is displayed on Figure 3. The iterations of the forward mode to compute the derivative with respect to x read

$$\begin{aligned}
&\frac{\partial x}{\partial x} = 1, \quad \frac{\partial y}{\partial x} = 0 \\
&\frac{\partial a}{\partial x} = \left[\frac{\partial a}{\partial x}\right]\frac{\partial x}{\partial x} = \frac{1}{x}\frac{\partial x}{\partial x} && \{x \mapsto a = \log(x)\} \\
&\frac{\partial b}{\partial x} = \left[\frac{\partial b}{\partial a}\right]\frac{\partial a}{\partial x} + \left[\frac{\partial b}{\partial y}\right]\frac{\partial y}{\partial x} = y\frac{\partial a}{\partial x} + 0 && \{(y,a) \mapsto b = ya\} \\
&\frac{\partial c}{\partial x} = \left[\frac{\partial c}{\partial b}\right]\frac{\partial b}{\partial x} = \frac{1}{2\sqrt{b}}\frac{\partial b}{\partial x} && \{b \mapsto c = \sqrt{b}\} \\
&\frac{\partial f}{\partial x} = \left[\frac{\partial f}{\partial b}\right]\frac{\partial b}{\partial x} + \left[\frac{\partial f}{\partial c}\right]\frac{\partial c}{\partial x} = 1\frac{\partial b}{\partial x} + 1\frac{\partial c}{\partial x} && \{(b,c) \mapsto f = b + c\}
\end{aligned}$$

One needs to run another forward pass to compute the derivative with respect to y

$$\begin{aligned}
&\frac{\partial x}{\partial y} = 0, \quad \frac{\partial y}{\partial y} = 1 \\
&\frac{\partial a}{\partial y} = \left[\frac{\partial a}{\partial x}\right]\frac{\partial x}{\partial y} = 0 && \{x \mapsto a = \log(x)\} \\
&\frac{\partial b}{\partial y} = \left[\frac{\partial b}{\partial a}\right]\frac{\partial a}{\partial y} + \left[\frac{\partial b}{\partial y}\right]\frac{\partial y}{\partial y} = 0 + a\frac{\partial y}{\partial y} && \{(y,a) \mapsto b = ya\} \\
&\frac{\partial c}{\partial y} = \left[\frac{\partial c}{\partial b}\right]\frac{\partial b}{\partial y} = \frac{1}{2\sqrt{b}}\frac{\partial b}{\partial y} && \{b \mapsto c = \sqrt{b}\} \\
&\frac{\partial f}{\partial y} = \left[\frac{\partial f}{\partial b}\right]\frac{\partial b}{\partial y} + \left[\frac{\partial f}{\partial c}\right]\frac{\partial c}{\partial y} = 1\frac{\partial b}{\partial y} + 1\frac{\partial c}{\partial y} && \{(b,c) \mapsto f = b + c\}
\end{aligned}$$

Dual numbers. A convenient way to implement this forward pass is to make use of so called "dual number", which is an algebra over the real where the number have the form $x + \varepsilon x'$ where ε is a symbol

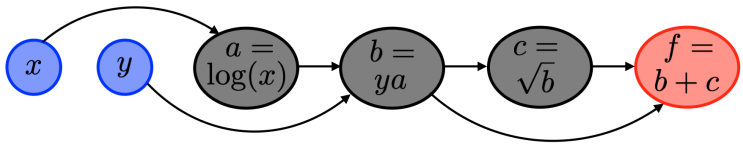

Figure 3: Example of a simple computational graph.

obeying the rule that $\varepsilon^2 = 0$. Here $(x, x') \in \mathbb{R}^2$ and x' is intended to store a derivative with respect to some input variable. These number thus obeys the following arithmetic operations

$$(x + \varepsilon x')(y + \varepsilon y') = xy + \varepsilon(xy' + yx') \quad \text{and} \quad \frac{1}{x + \varepsilon x'} = \frac{1}{x} - \varepsilon \frac{x'}{x^2}.$$

If f is a polynomial or a rational function, from these rules one has that

$$f(x + \varepsilon) = f(x) + \varepsilon f'(x).$$

For a more general basic function f, one needs to overload it so that

$$f(x + \varepsilon x') \overset{\text{def.}}{=} f(x) + \varepsilon f'(x) x'.$$

Using this definition, one has that

$$(f \circ g)(x + \varepsilon) = f(g(x)) + \varepsilon f'(g(x)) g'(x)$$

which corresponds to the usual chain rule. More generally, if $f(x_1, \ldots, x_s)$ is a function implemented using these overloaded basic functions, one has

$$f(x_1 + \varepsilon, x_2, \ldots, x_s) = f(x_1, \ldots, x_s) + \varepsilon \frac{\partial f}{\partial x_1}(x_1, \ldots, x_s)$$

and this evaluation is equivalent to applying the forward mode of automatic differentiation to compute $\frac{\partial f}{\partial x_1}(x_1, \ldots, x_s)$ (and similarly for the other variables).

4 Reverse Mode of Automatic Differentiation

Instead of evaluating the differentials $\frac{\partial x_k}{\partial x_1}$ which is problematic for a large p, the reverse mode evaluates the differentials $\frac{\partial x_t}{\partial x_k}$, i.e. it computes the derivative of the output node with respect to the all the inner nodes.

The method initialize the derivative of the final node

$$\frac{\partial x_t}{\partial x_t} = \mathrm{Id}_{n_t \times n_t},$$

and then iteratively makes use, from the last node to the first, of the following recursion formula

$$\forall\, k = t-1, t-2, \ldots, 1, \quad \frac{\partial x_t}{\partial x_k} = \sum_{m \in \mathrm{son}(k)} \frac{\partial x_t}{\partial x_m} \times \left[\frac{\partial x_m}{\partial x_k}\right] = \sum_{m \in \mathrm{son}(k)} \frac{\partial x_t}{\partial x_m} \times \frac{\partial f_m(x_1, \ldots, x_m)}{\partial x_k}.$$

The notation "parent(k)" denotes the nodes $\ell < k$ of the graph that are connected to k, see Figure 2, right.

Back-propagation. In the special case where $x_t \in \mathbb{R}$, then $\frac{\partial x_t}{\partial x_k} = [\nabla_{x_k} f(x)]^\top \in \mathbb{R}^{1\times n_k}$ and one can write the recursion on the gradient vector as follow

$$\forall\, k = t-1, t-2, \ldots, 1, \quad \nabla_{x_k} f(x) = \sum_{m \in \mathrm{son}(k)} \left(\frac{\partial f_m(x_1, \ldots, x_m)}{\partial x_k} \right)^\top (\nabla_{x_m} f(x)) \,.$$

where $\left(\frac{\partial f_m(x_1,\ldots,x_m)}{\partial x_k}\right)^\top \in \mathbb{R}^{n_k \times n_m}$ is the adjoint of the Jacobian of f_m. This form of recursion using adjoint is often referred to as "back-propagation", and is the most frequent setting in applications to ML.

In general, when $n_t = 1$, the backward is the optimal way to compute the gradient of a function. Its drawback is that it necessitate the pre-computation of all the intermediate variables $(x_k)_{k=p}^t$, which can be prohibitive in term of memory usage when t is large. There exists check-pointing method to alleviate this issue, but it is out of the scope of this course.

Simple example. We consider once again the fonction $f(x)$ of (1), the iterations of the reverse mode read

$$\begin{aligned}
&\frac{\partial f}{\partial f} = 1 \\
&\frac{\partial f}{\partial c} = \frac{\partial f}{\partial f}\left[\frac{\partial f}{\partial c}\right] = \frac{\partial f}{\partial f} 1 && \{c \mapsto f = b + c\} \\
&\frac{\partial f}{\partial b} = \frac{\partial f}{\partial c}\left[\frac{\partial c}{\partial b}\right] + \frac{\partial f}{\partial f}\left[\frac{\partial f}{\partial b}\right] = \frac{\partial f}{\partial c}\frac{1}{2\sqrt{b}} + \frac{\partial f}{\partial f} 1 && \{b \mapsto c = \sqrt{b}, b \mapsto f = b + c\} \\
&\frac{\partial f}{\partial a} = \frac{\partial f}{\partial b}\left[\frac{\partial b}{\partial a}\right] = \frac{\partial f}{\partial b} y && \{a \mapsto b = ya\} \\
&\frac{\partial f}{\partial y} = \frac{\partial f}{\partial b}\left[\frac{\partial b}{\partial y}\right] = \frac{\partial f}{\partial b} a && \{y \mapsto b = ya\} \\
&\frac{\partial f}{\partial x} = \frac{\partial f}{\partial a}\left[\frac{\partial a}{\partial x}\right] = \frac{\partial f}{\partial a}\frac{1}{x} && \{x \mapsto a = \log(x)\}
\end{aligned}$$

The advantage of the reverse mode is that a single traversal of the computational graph allows to compute both derivatives with respect to x, y, while the forward more necessitates two passes.

5 Feed-forward Compositions

The simplest computational graphs are purely feedforward, and corresponds to the computation of

$$f = f_t \circ f_{t-1} \circ \ldots \circ f_2 \circ f_1 \tag{2}$$

for functions $f_k : \mathbb{R}^{n_{k-1}} \to \mathbb{R}^{n_k}$.

The forward function evaluation algorithm initializes $x_0 = x \in \mathbb{R}^{n_0}$ and then computes

$$\forall\, k = 1, \ldots, t, \quad x_k = f_k(x_{k-1})$$

where at the output, one retrieves $f(x) = x_t$.

Denoting $A_k \overset{\text{def.}}{=} \partial f_k(x_{k-1}) \in \mathbb{R}^{n_k \times n_{k-1}}$ the Jacobian, one has

$$\partial f(x) = A_t \times A_{t-1} \times \ldots A_2 \times A_1.$$

The forward (resp. backward) mode corresponds to the computation of the product of the Jacobian from right to left (resp. left to right)

$$\begin{aligned}
\partial f(x) &= A_t \times (A_{t-1} \times (\ldots \times (A_3 \times (A_2 \times A_1)))) \,, \\
\partial f(x) &= ((((A_t \times A_{t-1}) \times A_{t-2}) \times \ldots) \times A_2) \times A_1.
\end{aligned}$$

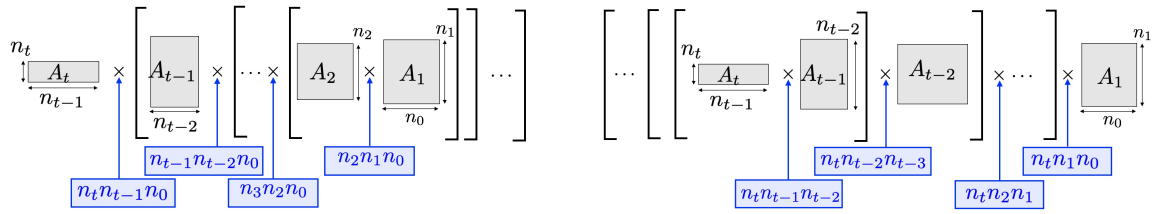

Figure 4: Complexity of forward (left) and backward (right) modes for composition of functions.

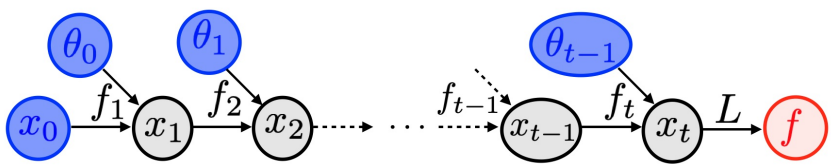

Figure 5: Computational graph for a feedforward architecture.

We note that the computation of the product $A \times B$ of $A \in \mathbb{R}^{n \times p}$ with $B \in \mathbb{R}^{p \times q}$ necessitates npq operations. As shown on Figure 4, the complexity of the forward and backward modes are

$$n_0 \sum_{k=1}^{t-1} n_k n_{k+1} \quad \text{and} \quad n_t \sum_{k=0}^{t-2} n_k n_{k+1}$$

So if $n_t \ll n_0$ (which is the typical case in ML scenario where $n_t = 1$) then the backward mode is cheaper.

6 Feed-forward Architecture

We can generalize the previous example to account for feed-forward architectures, such as neural networks, which are of the form

$$\forall\, k = 1, \ldots, t, \quad x_k = f_k(x_{k-1}, \theta_{k-1}) \tag{3}$$

where θ_{k-1} is a vector of parameters and $x_0 \in \mathbb{R}^{n_0}$ is given. The function to minimize has the form

$$f(\theta) \overset{\text{def.}}{=} L(x_t) \tag{4}$$

where $L : \mathbb{R}^{n_t} \to \mathbb{R}$ is some loss function (for instance a least square or logistic prediction risk) and $\theta = (\theta_k)_{k=0}^{t-1}$. Figure 5, top, displays the associated computational graph.

One can use the reverse mode automatic differentiation to compute the gradient of f by computing successively the gradient with respect to all (x_k, θ_k). One initializes

$$\nabla_{x_t} f = \nabla L(x_t)$$

and then recurse from $k = t-1$ to 0

$$z_{k-1} = [\partial_x f_k(x_{k-1}, \theta_{k-1})]^\top z_k \quad \text{and} \quad \nabla_{\theta_{k-1}} f = [\partial_\theta f_k(x_{k-1}, \theta_{k-1})]^\top (\nabla_{x_k} f) \tag{5}$$

where we denoted $z_k \overset{\text{def.}}{=} \nabla_{x_k} f(\theta)$ the gradient with respect to x_k.

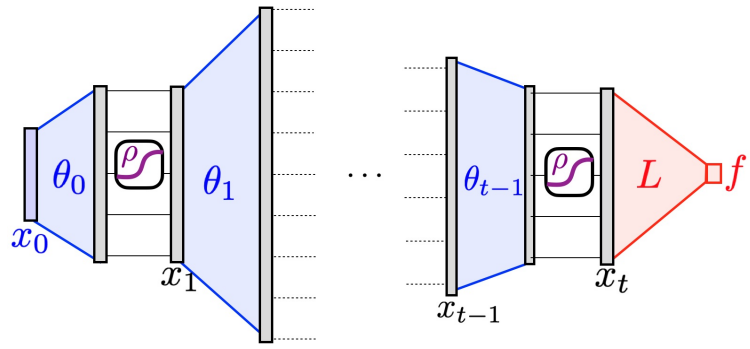

Figure 6: Multi-layer perceptron parameterization.

Multilayers perceptron. For instance, feedforward deep network (fully connected for simplicity) corresponds to using

$$\forall\, x_{k-1} \in \mathbb{R}^{n_{k-1}}, \quad f_k(x_{k-1}, \theta_{k-1}) = \rho(\theta_{k-1} x_{k-1}) \tag{6}$$

where $\theta_{k-1} \in \mathbb{R}^{n_k \times n_{k-1}}$ are the neuron's weights and ρ a fixed pointwise linearity, see Figure 6. One has, for a vector $z_k \in \mathbb{R}^{n_k}$ (typically equal to $\nabla_{x_k} f$)

$$\begin{cases} [\partial_x f_k(x_{k-1}, \theta_{k-1})]^\top(z_k) = \theta_{k-1}^\top w_k z_k, \\ [\partial_\theta f_k(x_{k-1}, \theta_{k-1})]^\top(z_k) = w_k x_{k-1}^\top \end{cases} \quad \text{where} \quad w_k \overset{\text{def.}}{=} \operatorname{diag}(\rho'(\theta_{k-1} x_{k-1})).$$

Link with adjoint state method. One can interpret (3) as a time discretization of a continuous ODE. One imposes that the dimension $n_k = n$ is fixed, and denotes $x(t) \in \mathbb{R}^n$ a continuous time evolution, so that $x_k \to x(k\tau)$ when $k \to +\infty$ and $k\tau \to t$. Imposing then the structure

$$f_k(x_{k-1}, \theta_{k-1}) = x_{k-1} + \tau u(x_{k-1}, \theta_{k-1}, k\tau) \tag{7}$$

where $u(x, \theta, t) \in \mathbb{R}^n$ is a parameterized vector field, as $\tau \to 0$, one obtains the non-linear ODE

$$\dot{x}(t) = u(x(t), \theta(t), t) \tag{8}$$

with $x(t = 0) = x_0$.

Denoting $z(t) = \nabla_{x(t)} f(\theta)$ the "adjoint" vector field, the discrete equations (10) becomes the so-called adjoint equations, which is a linear ODE

$$\dot{z}(t) = -[\partial_x u(x(t), \theta(t), t)]^\top z(t) \quad \text{and} \quad \nabla_{\theta(t)} f(\theta) = [\partial_\theta u(x(t), \theta(t), t)]^\top z(t).$$

Note that the correct normalization is $\frac{1}{\tau} \nabla_{\theta_{k-1}} f \to \nabla_{\theta(t)} f(\theta)$

7 Recurrent Architectures

Parametric recurrent functions are obtained by using the same parameter $\theta = \theta_k$ and $f_k = h$ recursively in (6), so that

$$\forall\, k = 1, \ldots, t, \quad x_k = h(x_{k-1}, \theta). \tag{9}$$

We consider a real valued function of the form

$$f(\theta) = L(x_t, \theta)$$

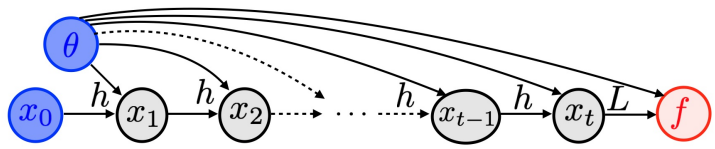

Figure 7: Computational graph for a recurrent architecture.

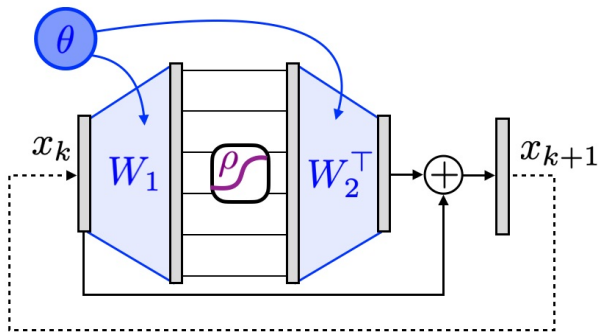

Figure 8: Recurrent residual perceptron parameterization.

so that here the final loss depends on θ (which is thus more general than (4)). Figure 7, bottom, displays the associated computational graph.

The back-propagation then operates as

$$\nabla_{x_{k-1}} f = [\partial_x h(x_{k-1}, \theta)]^\top \nabla_{x_k} f \quad \text{and} \quad \nabla_\theta f = \nabla_\theta L(x_t, \theta) + \sum_k [\partial_\theta h(x_{k-1}, \theta)]^\top \nabla_{x_k} f. \tag{10}$$

Similarly, writing $h(x, \theta) = x + \tau u(x, \theta)$, letting $(k, k\tau) \to (+\infty, t)$, one obtains the forward non-linear ODE with a time-stationary vector field

$$\dot{x}(t) = u(x(t), \theta)$$

and the following linear backward adjoint equation, for $f(\theta) = L(x(T), \theta)$

$$\dot{z}(t) = -[\partial_x u(x(t), \theta)]^\top z(t) \quad \text{and} \quad \nabla_\theta f(\theta) = \nabla_\theta L(x(T), \theta) + \int_0^T [\partial_\theta f(x(t), \theta)]^\top z(t) \mathrm{d}t. \tag{11}$$

with $z(0) = \nabla_x L(x_t, \theta)$.

Residual recurrent networks. A recurrent network is defined using

$$h(x, \theta) = x + W_2^\top \rho(W_1 x)$$

as displayed on Figure 8, where $\theta = (W_1, W_2) \in (\mathbb{R}^{n \times q})^2$ are the weights and ρ is a pointwise non-linearity. The number q of hidden neurons can be increased to approximate more complex functions. In the special case where $W_2 = -\tau W_1$, and $\rho = \psi'$, then this is a special case of an argmin layer (13) to minimize the function $\mathcal{E}(x, \theta) = \psi(W_1 x)$ using gradient descent, where $\psi(u) = \sum_i \psi(u_i)$ is a separable function. The Jacobians $\partial_\theta h$ and $\partial_x h$ are computed similarly to (11).

Mitigating memory requirement. The main issue of applying this backpropagation method to compute $\nabla f(\theta)$ is that it requires a large memory to store all the iterates $(x_k)_{k=0}^t$. A workaround is to use checkpointing, which stores some of these intermediate results and re-run partially the forward algorithm to

reconstruct missing values during the backward pass. Clever hierarchical method perform this recursively in order to only require $\log(t)$ stored values and a $\log(t)$ increase on the numerical complexity.

In some situation, it is possible to avoid the storage of the forward result, if one assume that the algorithm can be run backward. This means that there exists some functions g_k so that

$$x_k = g_k(x_{k+1}, \ldots, x_t).$$

In practice, this function typically also depends on a few extra variables, in particular on the input values $(x_0, \ldots, x_s)$.

An example of this situation is when one can split the (continuous time) variable as $x(t) = (r(t), s(t))$ and the vector field u in the continuous ODE (8) has a symplectic structure of the form $u((r,s),\theta,t) = (F(s,\theta,t), G(r,\theta,t))$. One can then use a leapfrog integration scheme, which defines

$$r_{k+1} = r_k + \tau F(s_k, \theta_k, \tau k) \quad \text{and} \quad s_{k+1} = s_k + \tau G(r_{k+1}, \theta_{k+1/2}, \tau(k+1/2)).$$

One can reverse these equation exactly as

$$s_k = s_{k+1} - \tau G(r_{k+1}, \theta_{k+1/2}, \tau(k+1/2)). \quad \text{and} \quad r_k = r_{k+1} - \tau F(s_k, \theta_k, \tau k).$$

Fixed point maps In some applications (some of which are detailed bellow), the iterations x_k converges to some $x^\star(\theta)$ which is thus a fixed point

$$x^\star(\theta) = h(x^\star(\theta), \theta).$$

Instead of applying the back-propagation to compute the gradient of $f(\theta) = L(x_t, \theta)$, one can thus apply the implicit function theorem to compute the gradient of $f^\star(\theta) = L(x^\star(\theta), \theta)$. Indeed, one has

$$\nabla f^\star(\theta) = [\partial x^\star(\theta)]^\top (\nabla_x L(x^\star(\theta), \theta)) + \nabla_\theta L(x^\star(\theta), \theta). \tag{12}$$

Using the implicit function theorem one can compute the Jacobian as

$$\partial x^\star(\theta) = -\left(\frac{\partial h}{\partial x}(x^\star(\theta), \theta)\right)^{-1} \frac{\partial h}{\partial \theta}(x^\star(\theta), \theta).$$

In practice, one replace in these formulas $x^\star(\theta)$ by x_t, which produces an approximation of $\nabla f(\theta)$. The disadvantage of this method is that it requires the resolution of a linear system, but its advantage is that it bypass the memory storage issue of the backpropagation algorithm.

Argmin layers One can define a mapping from some parameter θ to a point $x(\theta)$ by solving a parametric optimization problem

$$x(\theta) = \operatorname*{argmin}_x \mathcal{E}(x, \theta).$$

The simplest approach to solve this problem is to use a gradient descent scheme, $x_0 = 0$ and

$$x_{k+1} = x_k - \tau \nabla \mathcal{E}(x_k, \theta). \tag{13}$$

This has the form (7) when using the vector field $u(x,\theta) = \nabla \mathcal{E}(x_k, \theta)$.

Using formula (12) in this case where $h = \nabla \mathcal{E}$, one obtains

$$\nabla f^\star(\theta) = -\left(\frac{\partial^2 \mathcal{E}}{\partial x \partial \theta}(x^\star(\theta), \theta)\right)^\top \left(\frac{\partial^2 \mathcal{E}}{\partial x^2}(x^\star(\theta), \theta)\right)^{-1} (\nabla_x L(x^\star(\theta), \theta)) + \nabla_\theta L(x^\star(\theta), \theta)$$

In the special case where the function $f(\theta)$ is the minimized function itself, i.e. $f(\theta) = \mathcal{E}(x^\star(\theta), \theta)$, i.e. $L = \mathcal{E}$, then one can apply the implicit function theorem formula (12), which is much simpler since in this case $\nabla_x L(x^\star(\theta), \theta) = 0$ so that

$$\nabla f^\star(\theta) = \nabla_\theta L(x^\star(\theta), \theta). \tag{14}$$

This result is often called Danskin theorem or the envelope theorem.

Sinkhorn's algorithm Sinkhorn algorithm approximates the optimal distance between two histograms $a \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ using the following recursion on multipliers, initialized as $x_0 \stackrel{\text{def.}}{=} (u_0, v_0) = (1_n, 1_m)$

$$u_{k+1} = \frac{a}{K v_k}, \quad \text{and} \quad v_{k+1} = \frac{b}{K^\top u_k}.$$

where $\frac{\cdot}{\cdot}$ is the pointwise division and $K \in \mathbb{R}_+^{n \times m}$ is a kernel. Denoting $\theta = (a, b) \in \mathbb{R}^{n+m}$ and $x_k = (u_k, v_k) \in \mathbb{R}^{n+m}$, the OT distance is then approximately equal to

$$f(\theta) = \mathcal{E}(x_t, \theta) \stackrel{\text{def.}}{=} \langle a, \log(u_t) \rangle + \langle b, \log(v_t) \rangle - \varepsilon \langle K u_t, v_t \rangle.$$

Sinkhorn iteration are alternate minimization to find a minimizer of $\mathcal{E}$.

Denoting $\mathcal{K} \stackrel{\text{def.}}{=} \begin{pmatrix} 0 & K \\ K^\top & 0 \end{pmatrix} \in \mathbb{R}^{(n+m) \times (n+m)}$, one can re-write these iterations in the form (9) using

$$h(x, \theta) = \frac{\theta}{\mathcal{K} x} \quad \text{and} \quad L(x_t, \theta) = \mathcal{E}(x_t, \theta) = \langle \theta, \log(x_t) \rangle - \varepsilon \langle K u_t, v_t \rangle.$$

One has the following differential operator

$$[\partial_x h(x, \theta)]^\top = -\mathcal{K}^\top \operatorname{diag}\left(\frac{\theta}{(\mathcal{K} x)^2} \right), \quad [\partial_\theta h(x, \theta)]^\top = \operatorname{diag}\left(\frac{1}{\mathcal{K} x} \right).$$

Similarly as for the argmin layer, at convergence $x_k \to x^\star(\theta)$, one finds a minimizer of $\mathcal{E}$, so that $\nabla_x L(x^\star(\theta), \theta) = 0$ and thus the gradient of $f^\star(\theta) = \mathcal{E}(x^\star(\theta), \theta)$ can be computed using (14) i.e.

$$\nabla f^\star(\theta) = \log(x^\star(\theta)).$$