# THE ADJOINT TRICK

ANTOINE LEVITT

## 1. Notations and finite differences

This short note is to introduce a trick variously known as "reverse/backard-mode automatic differentiation" (computer science), "backpropagation" (machine learning), "adjoint method" (differential equations), and variations around the word "Lagrangian" (quantum chemistry). The emphasis is on efficient methods for gradients, and linking the "tricks of the trade" used in different scientific communities with the general methodology of automatic differentiation.

Imagine we have a differentiable function $F : \mathbb{R}^{N_{\text{in}}} \to \mathbb{R}^{N_{\text{out}}}$ whose derivatives we wish to compute automatically. At a point $x \in \mathbb{R}^{N_{\text{in}}}$, we will note $dF_x$ the differential, a map from $\mathbb{R}^{N_{\text{in}}}$ to $\mathbb{R}^{N_{\text{out}}}$, which can be identified by its Jacobian $J_{F,x}$, a $N_{\text{out}} \times N_{\text{in}}$ matrix, with

$$J_{F,x}[i,j] = \langle e_i, dF_x \cdot e_j \rangle \,.$$

In the case where $N_{\text{out}} = 1$, we will write $\nabla F(x) = (J_{F,x})^T$ for the gradient of $F$ at $x$, identifying $N_{\text{in}} \times 1$ matrices with vectors. We will in the following drop the $x$ index, and write $J_F$, $dF$, $\nabla F$. We recall that, if $L : \mathbb{R}^{N_{\text{in}}} \to \mathbb{R}^{N_{\text{out}}}$ is a linear map, its adjoint $L^*$ is characterized by $\langle L^* x, y \rangle = \langle x, Ly \rangle$. If $L$ is represented by the matrix $A$, $L^*$ is represented by the matrix $A^T$.

The first reflex is finite differences:

$$J_F[:,j] \approx \frac{F(x + \alpha e_j) - F(x)}{\alpha}$$

with a small $\alpha$. This is however imprecise. Assume that $F$ is only known up to an error of $\varepsilon_F$ (e.g. coming from machine arithmetic, with $\varepsilon_F \approx 10^{-16}$ in double precision), and that $x$, $F$ and $F'$ are of order 1. Then $(F(x + \varepsilon e_j) - F(x))/\alpha$ is in error by about $\varepsilon_F/\alpha$. On the other hand, the error made by replacing the derivative by its finite-difference approximation is about $\alpha$. Balancing the two terms leads to $\alpha = \sqrt{\varepsilon_F}$, with an error of $\sqrt{\varepsilon_F}$. Using higher-order finite differences improves this power, but is more expensive.

## 2. Forward mode

Forward-mode automatic differentiation is a way to compute derivatives *exactly* (meaning, up to machine precision). It works by following the flow of the code. At its most basic level, a program to compute $F$ is a sequence of transformations $F_1 \to F_2 \to \cdots \to F_I$, where $F_i : \mathbb{R}^{N_i} \to \mathbb{R}^{N_{i+1}}$, and $N_0 = N_{\text{in}}, N_{I+1} = N_{\text{out}}$. We then have $F(x_1) = F_I(\dots(F_2(F_1(x_1))))$. The chain rule is

$$dF \cdot \delta x_1 = dF_I \cdot (\dots dF_2 \cdot (dF_1 \cdot \delta x_1)) \tag{1}$$

We can implement this by keeping track of $\delta x_i$ along the computation. We first compute $x_2 = F_1(x_1)$ and set $\delta x_2 = dF_1 \cdot \delta x_1$. Then we compute $x_3 = F_2(x_2)$ and set $\delta x_3 = dF_2 \cdot \delta x_2$. We iterate until we have computed $\delta x_{I+1} = dF \cdot x_1$.

This can be interpreted as propagating a perturbation: $\delta x_i$ is the perturbation of $x_i$ caused by the initial perturbation $\delta x_1$ on $x_1$. This is conveniently implemented using dual numbers: instead of storing a number (or array) $x$, we store the pair $(x, \delta x)$. Operations can be defined on dual numbers: for instance, if $x$ is a scalar, $(x, \delta x) * (y, \delta y) = (x * y, x * \delta y + \delta x * y)$.

## 3. Reverse mode

So far, so boring. The above procedure is exact, so we have managed to compute derivatives more accurately. But imagine now we want to compute a gradient, i.e. that $N_{I+1} = N_{\text{out}} = 1$. Finite differences require $N_{\text{in}}$ applications of $F$. With forward-mode automatic differentiation described above, the procedure needs to be applied for all input perturbations $\delta x_1 = e_1, \dots, e_{N_{\text{in}}}$. This should make the following extremely surprising

*Date*: March 6, 2020.

**Claim.** *Let $F : \mathbb{R}^N \to \mathbb{R}$ be a differentiable function. The cost of computing the gradient $\nabla F$ is asymptotically no greater than that of computing $F$.*

By asymptotically we mean that computing $\nabla F$ can be slower than computing $F$ by a constant factor, but that constant factor should not depend on $N$. This means that the forward method and finite differences are both asymptotically suboptimal. How can this be?

Note that computing gradients by hand is full of tricks, that are applied manually and often feel a bit magical. For instance, imagine that $F(x) = \langle b, Ax\rangle$ where $b$ is a given vector and $A$ a sparse matrix. The naive procedure would take for $x$ the unit vectors in order, effectively computing the full matrix $A$. However, $F(x) = \langle A^T b, x\rangle$ and so the gradient is easily computed: it's $A^T b$, which can be computed very efficiently (in about the same time as $F$). More generally, if $F(x) = F_2(F_1(x))$ with $F_1 : \mathbb{R}^{N_{\text{in}}} \to \mathbb{R}^{N_{\text{intermediary}}}$ and $F_2 : \mathbb{R}^{N_{\text{intermediary}}} \to \mathbb{R}$, then one can compute $\nabla F = (J_{F_1})^T \nabla F_2$. In this formulation, only one intermediary vector ($\nabla F_2$) is used: the perturbation of the intermediate vectors with respect to all the possible inputs is not needed.

Reverse-mode is a generalization of this idea. It is based on a simplistic but deep trick: taking the adjoint of (1), to get

$$(dF)^* \cdot \text{ƍ}x_{I+1} = (dF_1)^* \cdot ((dF_2)^* \cdot (\dots (dF_I)^* \cdot \text{ƍ}x_{I+1}) \tag{2}$$

Given a $\text{ƍ}x_{I+1} \in \mathbb{R}^{N_{\text{out}}}$, we compute $\text{ƍ}x_I = (dF_I)^* \cdot \text{ƍ}x_{I+1}$. Then, we iterate until we get $\text{ƍ}x_1 = (dF)^* \cdot \text{ƍ}x_{I+1}$.

The great advantage of using this formulation is that to know $(dF)^*$ completely, $\text{ƍ}x_{I+1}$ only needs to span $\mathbb{R}^{N_{\text{out}}}$. Therefore, when $N_{\text{out}} = 1$ (and so we want to compute the gradient $\nabla F$ representing the linear operator $(dF)^*$), only one perturbation ($\text{ƍ}x_{I+1} = 1$) is needed. This leads to the claim above.

The reverse delta $\text{ƍ}x$ (sometimes noted $\overline{x}$ in the AD literature) is there to emphasize the dual status of $\text{ƍ}x$ compared to the perturbation $\delta x$. The role of $\text{ƍ}x_i$ is to answer the following question: assume the computation was stopped at step $i$, and taken up again, with as input a perturbation of $x_i$. What is the gradient of $F$ with respect to this perturbation? Unlike $\delta x_i$, which represented the perturbation to $x_i$ caused by the input $\delta x_1$, here the chain of causality is reversed. Knowing the gradient at step $i$ does not help to compute the gradient at step $i+1$; rather, the opposite is true: if we know the gradient $\text{ƍ}x_{i+1}$ of $F$ with respect to $x_{i+1}$, we can figure out the gradient $\text{ƍ}x_i$ of $F$ with respect to $x_i$ by simply acting with $dF_i^*$.

This is very counter-intuitive because of the reversal of causality, which we are most used to see in the context of inverses. Computing $\text{ƍ}x_i = dF_i^* \text{ƍ}x_{i+1}$ has nothing to do with inverting $dF_i$. For instance, when $dF_i = 0$ ($x_{i+1}$ is not affected by $x_i$), then $\text{ƍ}x_i = 0$ (the final output $F$ is not affected by $x_i$). $\text{ƍ}x_i$ can be thought of as the sensitivity of the output $F(x)$ with respect to $x_i$; $\delta x_i$ can be thought of as the sensitivity of $x_i$ with respect to the input $x_1$.

The differential $dF$ takes the sensitivity of the input of $F$ to a parameter, and yields the sensitivity of the output to that parameter.
The adjoint differential $dF^*$ takes the sensitivity of a parameter to the output of $F$, and yields the sensitivity of that parameter to the input.

This reversal of causality also has important consequences for implementation: the propagation of $\text{ƍ}x_i$ has to be done *in reverse* compared to the normal flow of the program. This means that every intermediate computations $x_i$ have to be stored. There are techniques to mitigate this, but the process is considerably less straightforward than forward-mode automatic differentiation. In practice, this can be helped by defining custom adjoints: defining “by hand” the action of $dF^*$ on a vector, without needing to go into the low-level details of how $F$ is computed

In order to not get lost, it is important to remember that, if $j > i$, then $\delta x_i$ causes $\delta x_j$, and $\text{ƍ}x_j$ causes $\text{ƍ}x_i$. A helpful way to organize computations is the simple-looking (and therefore important) equation

$$\langle \text{ƍ}x_i, \delta x_i\rangle = \langle dF_i^* \cdot \text{ƍ}x_{i+1}, \delta x_i\rangle = \langle \text{ƍ}x_{i+1}, dF_i \cdot \delta x_i\rangle = \langle \text{ƍ}x_{i+1}, \delta x_{i+1}\rangle \tag{3}$$

This is helpful in the following way: given a perturbation $\delta x_i$, it is usually straightforward to propagate it to $\delta x_{i+1} = dF_i \cdot \delta x_i$. Then, using (3), given $\text{ƍ}x_{i+1}$, one gets the action of $\text{ƍ}x_i$ on any vector $\delta x_i$, which then determines $\text{ƍ}x_i$ uniquely.

## 4. Examples and applications

4.1. **Direct adjoints.** A direct extension of the trick above example for the gradient of $F(x) = \langle b, Ax\rangle$ is the “adjoint equation” for computing gradients of objective functions that depend on the solution of a linear equation: if $Lu = f$ where $u, f \in \mathbb{R}^N$, then what is the gradient of $F(f) = \langle b, u\rangle$ with respect to $f$? Of course, $F(f) = \langle b, u\rangle = \langle b, L^{-1}f\rangle = \langle (L^{-1})^T b, f\rangle$ and so the gradient is $L^{-1})^T b$. Let us see how we can get that result as an application of the theory above.

The flow of computation is $f \to u \to F$, from which we obtain $\delta f \to \delta u \to \delta F$: $\delta u = L^{-1}\delta f$, and $\delta F = \langle b, \delta u\rangle_{\mathbb{R}^N}$. Now we need to reverse the flow: $ƃF \to ƃu \to ƃf$. We use

$$\langle ƃu, \delta u\rangle_{\mathbb{R}^N} = \langle ƃF, \delta F\rangle_{\mathbb{R}} = \langle ƃF, \langle b, \delta u\rangle_{\mathbb{R}^N}\rangle_{\mathbb{R}} = \langle ƃF b, \delta u\rangle_{\mathbb{R}^N}$$

so $ƃu = ƃF b$. Then,

$$\langle ƃf, \delta f\rangle_{\mathbb{R}^N} = \langle ƃu, \delta u\rangle_{\mathbb{R}^N} = \langle ƃu, L^{-1}\delta f\rangle_{\mathbb{R}^N} = \langle (L^{-1})^T ƃu, \delta f\rangle_{\mathbb{R}^N}$$

so $ƃf = (L^{-1})^T ƃu$. It follows that $\nabla F = (L^T)^{-1} b$. This is also true when $u$ and $f$ are functions, and $L$ is an operator (the adjoint operator then being defined by $\int (L^T u)v = \int u(Lv)$ for all $v$).

This also works for nonlinear problems, by solving an equation with the adjoint Jacobian. For instance, what is the gradient of

$$F(A) = x^T B x, \text{ where}$$
$$Ax = \lambda x, \quad \|x\|^2 = 1,$$

assuming $\lambda$ is a simple eigenvalue of $A$, with $A$ and $B$ belonging to $M_{\mathrm{sym}}(N)$, the set of symmetric matrices size $N$? The Jacobian of the map $(x, \lambda) \mapsto (Ax - \lambda, \|x\|^2 = 1)$ is

$$J = \begin{pmatrix} A - \lambda & -x \\ 2x^T & 0 \end{pmatrix}$$

and so

$$\begin{pmatrix} \delta x \\ \delta\lambda \end{pmatrix} = -J^{-1} \begin{pmatrix} \delta A x \\ 0 \end{pmatrix}$$

which can be computed by a Schur complement as $\delta x = -(A-\lambda)^+ \delta A x$ with $(A-\lambda)^+$ the pseudo-inverse. Therefore, $\delta F = 2B\delta x = -2B(A-\lambda)^+\delta A x$.

We have now computed the forward differential, through the sequence $\delta A \to \delta x \to \delta F$. Now we go in reverse, ie $ƃF \to ƃx \to ƃA$. We use the relationship

$$\langle ƃx, \delta x\rangle_{\mathbb{R}^N} = \langle ƃF, \delta F\rangle_{\mathbb{R}} = \langle ƃF, \langle 2Bx, \delta x\rangle_{\mathbb{R}^N}\rangle_{\mathbb{R}} = \langle ƃF 2B^T x, \delta x\rangle_{\mathbb{R}^N}$$

for all $\delta x$ (which determines $\delta F$) and $ƃF$ (which determines $ƃx$) to deduce that

$$ƃx = ƃF 2B^T x.$$

Then we go further back in the same way (remember that $\langle A, B\rangle_{M_{\mathrm{sym}}(N)} = \mathrm{Tr}(AB)$):

$$\begin{aligned}\langle ƃA, \delta A\rangle_{M_{\mathrm{sym}}(N)} &= \langle ƃx, \delta x\rangle_{\mathbb{R}^N} = \langle ƃx, -(A-\lambda)^+\delta A x\rangle_{\mathbb{R}^N} = \langle -(A-\lambda)^+ ƃx, \delta A x\rangle_{\mathbb{R}^N} \\ &= \mathrm{T}r(-x((A-\lambda)^+ ƃx)^T \delta A) = -\left\langle \left(x((A-\lambda)^+ ƃx)^T + ((A-\lambda)^+ ƃx)x^T\right), \delta A\right\rangle_{M_{\mathrm{sym}}(N)}\end{aligned}$$

so

$$ƃA = -x((A-\lambda)^+ ƃx)^T - ((A-\lambda)^+ ƃx)x^T$$

We can then combine this with the equation for $ƃx$ to compute $ƃA$ as a function of $ƃF$, hence the gradient.

4.2. **Neural networks.** Until now we have applied the above procedure to only two levels. The simplest visualization of multi-level reverse-mode automatic differentiation is to neural networks. Simple neural networks are defined by functions $F_i(x_i) = f(W_i * x_i + b_i)$ where $f$ is a nonlinear activation function, $W_i$ is a matrix of weights and $b_i$ a vector of biases. The theory above applies straightforwardly to compute the gradient of $F$ (the loss) with respect to all the $W_i$ and $b_i$: $ƃW_i$ and $ƃb_i$. There, the implementation is completely straightforward: each neuron $i$ remembers their value $x_i$ during the evaluation of $F$ (the "forward pass"); then the adjoint equation is used to update the gradients of the loss function, starting from the last layer (the "backward pass").

4.3. **Time integration.** A continuous-time version of the neural network case is ODE solving: imagine we want to compute the gradient of $F_T(x_0) = b(x(T))$, where $\dot{x}(t) = f(t, x(t))$ and $b : \mathbb{R}^N \to \mathbb{R}$. The forward differential is

$$\delta F = dF_T(x_0) \cdot \delta x_0 = \nabla b(x(T)) \cdot \delta x(T),$$

where $\frac{d}{dt}\delta x(t) = \frac{\partial f}{\partial x}(t, x(t)) \cdot \delta x(t)$. Its adjoint is defined by

$$\begin{aligned}
\langle dF_T(x_0)^* \cdot \rotatebox[origin=c]{180}{$\delta$} F, \delta x_0 \rangle &= \langle \rotatebox[origin=c]{180}{$\delta$} F, dF_T(x_0) \cdot \delta x_0 \rangle \\
&= \langle \rotatebox[origin=c]{180}{$\delta$} F, \nabla b(x(T)) \cdot \delta x(T) \rangle \\
&= \langle \rotatebox[origin=c]{180}{$\delta$} F \nabla b(x(T)), \delta x(T) \rangle
\end{aligned}$$

This gives a way to compute individual components

$$\langle \rotatebox[origin=c]{180}{$\delta$} x_0, \delta x_0 \rangle = \langle dF_T(x_0)^* \cdot 1, \delta x_0 \rangle = \langle \nabla b(x(T)), \delta x(T) \rangle$$

of $\rotatebox[origin=c]{180}{$\delta$} x_0$ for all $\delta x_0$, by propagating $\delta x_0$ to $\delta x(T)$ (the equivalent of (1)), but that's not what we want: we want to compute it for all $\delta x_0$ in one go (the equivalent of (2)). So we need to reverse the flow of computation, and introduce the intermediary $\rotatebox[origin=c]{180}{$\delta$} x(t)$, which answers the following question: suppose we start the ODE at time $t$ with initial condition $x(t)$; what is the gradient of $F_T$ with respect to $x(t)$? Obviously $\rotatebox[origin=c]{180}{$\delta$} x(T) = \nabla b(x(T))$. But we also have for all $t$, $\delta x(t)$,

$$\langle \rotatebox[origin=c]{180}{$\delta$} x(t), \delta x(t) \rangle = \langle \rotatebox[origin=c]{180}{$\delta$} x(T), \delta x(T) \rangle$$

where $\delta x(t')$ still satisfies the ODE $\frac{d\delta x}{dt}(t') = \frac{\partial F_T}{\partial x} f(t', x(t')) \cdot \delta x(t')$ with initial condition $\delta x(t)$. Differentiating this relationship with respect to $t$, we get

$$\begin{aligned}
\left\langle \frac{d\rotatebox[origin=c]{180}{$\delta$} x}{dt}(t), \delta x(t) \right\rangle &= -\left\langle \rotatebox[origin=c]{180}{$\delta$} x(t), \frac{\partial f}{\partial x}(t, x(t)) \delta x(t) \right\rangle \\
&= -\left\langle \frac{\partial f}{\partial x}(t, x(t))^* \rotatebox[origin=c]{180}{$\delta$} x(t), \delta x(t) \right\rangle
\end{aligned}$$

from where it follows that

$$\frac{d}{dt} \rotatebox[origin=c]{180}{$\delta$} x(t) = \frac{\partial f}{\partial x}(t, x(t))^* \rotatebox[origin=c]{180}{$\delta$} x(t)$$

which is an ODE that can be solved backwards in time from $\rotatebox[origin=c]{180}{$\delta$} x(T)$ to get $\rotatebox[origin=c]{180}{$\delta$} x(0)$.