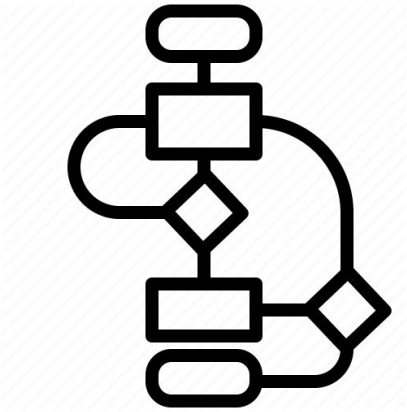


LINMA2472 - Algorithm in data science

Simon Desmidt

Academic year 2025-2026 - Q1

UCLouvain

Contents

Automatic differentiation

The Automatic differentiation is an algorithmic technique to compute automatically the derivative (gradient) of a function defined in a computer program. Unlike symbolic differentiation (done by hand) and numerical differentiation (finite difference approximation), automatic differentiation exploits the fact that every function can be decomposed into a sequence of elementary operations (addition, multiplication, sine, exponential, etc.) and so that we can apply the chain rule to compute the derivative of the whole function. Thus we can compute the gradient of a function exactly and efficiently.
Automatic differentiation is widely used in machine learning because for the neural networks, we need to compute the gradient of a loss function with respect to the parameters of the model (weights and biases) to update them during the training process and it would be difficult to compute this manually for each node.

1.1 Chain rule

There is two ways to apply the chain rule to compute the gradient of a function: forward differentiation and backward differentiation. Suppose that we have a composition of m functions. The chain rule gives us:

$$f'(x) = f'_m(f_{m-1}(f_{m-2}(...f_1(x)...))) \cdot ... \cdot f'_2(f_1(x)) \cdot f'_1(x) \tag{1.1}$$

Let's define:

$$\begin{cases} s_0 &= x \\ s_k &= f_k(s_{k-1}) \end{cases} \tag{1.2}$$

We thus get:

$$f'(x) = f'_m(s_{m-1}) \cdot ... \cdot f'_2(s_1) \cdot f'_1(s_0) \tag{1.3}$$

Based on this, we can define the forward and backward differentiation algorithms.

1.2 Forward differentiation

Also called forward mode, this algorithm consists in propagating forward the derivative and the values at the same time. It can be represented by this graph where the blue part represents the values and the green part the derivatives:

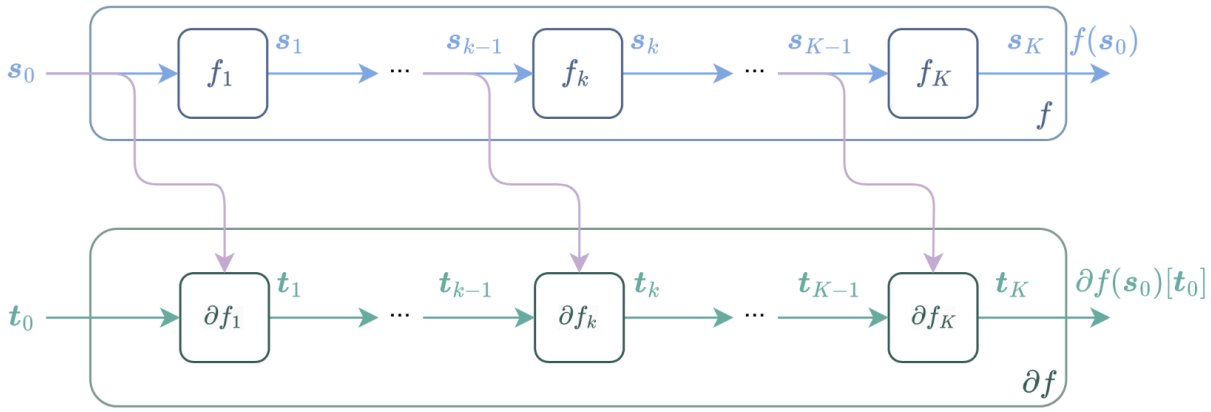


Figure 1.1: Forward differentiation

And it can be computed with the following recurrence relation:

$$\begin{cases} t_0 &= 1 \\ t_k &= f'_k(s_{k-1}) \cdot t_{k-1} \end{cases} \tag{1.4}$$

It is simple to implement and very efficient for functions with a small number of input variables. However, it becomes inefficient for functions with a large number of input variables because we need to compute the derivative for each input variable separately. So in practice for neural networks where we have a large number of input variables (weights and biases), we use the backward differentiation.

1.3 Backward differentiation

Also called backward mode, this algorithm consists in propagating the derivative backward and the values forward at the same time. It can be represented by this graph where the blue part represents the values and the orange part the derivatives:

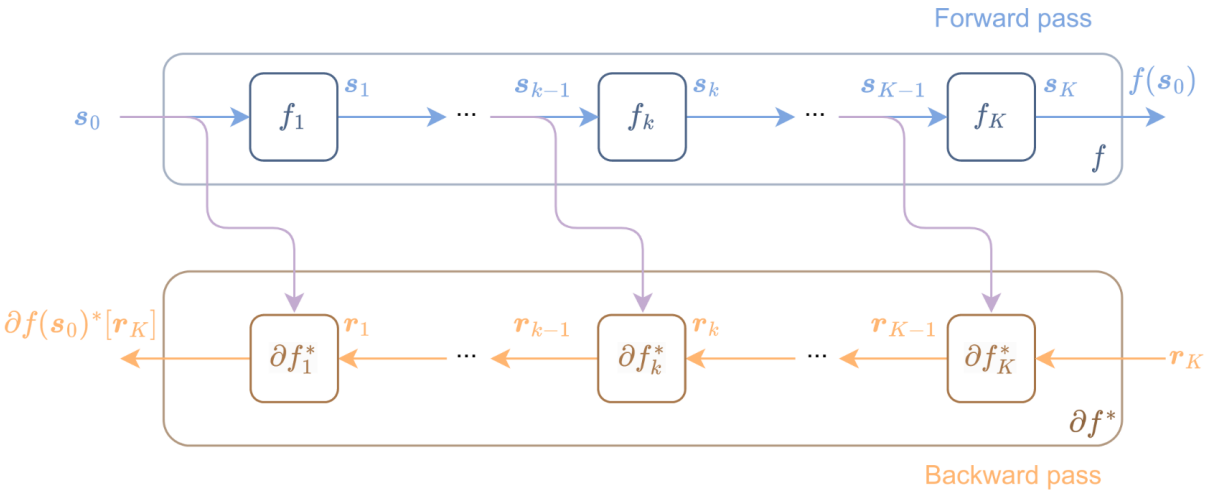


Figure 1.2: Backward differentiation

The idea is to compute all the intermediate values s_k in a forward pass and then compute the derivatives r_k based on the output in a backward pass. It can be computed

with the following recurrence relation:

$$\begin{cases} r_m & = 1 \\ r_k & = r_{k+1} \cdot f'_{k+1}(s_k) \end{cases} \tag{1.5}$$

This method is more complex to implement but it is very efficient for functions with a large number of input variables and a small number of output variables typically 1, the loss function.

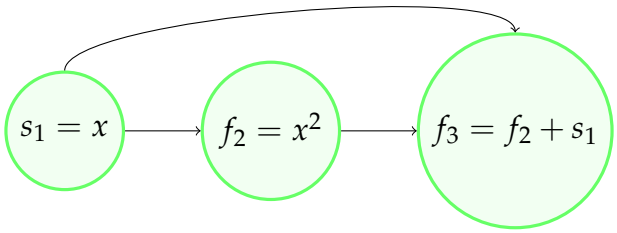
1.4 Computational graph and multivariate differentiation

1.4.1 Computational graph

To represent the computation of a function, we can use a computational graph. It is a directed acyclic graph where the nodes represent the operations and the edges represent the variables. For example, consider the function with $f_1(x) = x = s_1$ and $f_2(x) = x^2 = s_2$:

$$f_3(s_1, s_2) = s_1 + s_2 = x + x^2 \tag{1.6}$$

The computational graph is:



1.4.2 Multivariate differentiation

Let's consider the function of the computational graph above:

$$f_3(f_1(x), f_2(x)) = s_3 = f_1(x) + f_2(x) = s_1 + s_2 = x + x^2 \tag{1.7}$$

following the chain rule, we have:

$$f'_3(x) = \frac{\partial f_3}{\partial s_1}\frac{\partial s_1}{\partial x} + \frac{\partial f_3}{\partial s_2}\frac{\partial s_2}{\partial x} \tag{1.8}$$

For the forward automatic differentiation, we work the same way as before, we propagate the values and the derivatives forward. But when we have a node with multiple inputs, we need to use formula derivated from the chain rule. For the function f_3 that we want to evaluate in $x = 3$, we will have:

$$\begin{cases} t_0 & = 1 \\ t_1 & = f'_1(x)|_{x=3} \cdot t_0 = 1 \\ t_2 & = f'_2(x)|_{x=3} \cdot t_0 = 6 \\ t_3 & = \frac{\partial f_3}{\partial s_1}|_{x=3} \cdot t_1 + \frac{\partial f_3}{\partial s_2}|_{x=3} \cdot t_2 = 7 \end{cases} \tag{1.9}$$

For the backward automatic differentiation, first we need to initialize the gradient accumulator to 0.

$$\frac{\partial s_3}{\partial s_1} = \frac{\partial s_3}{\partial s_2} = \frac{\partial s_3}{\partial x} = 0 \tag{1.10}$$

Then we compute the intermediate values in a forward pass:

$$\begin{aligned} \frac{\partial s_3}{\partial s_1} += 1 \Rightarrow \frac{\partial s_3}{\partial x} += 1 \cdot 1|_{x=3} \\ \frac{\partial s_3}{\partial s_2} += 1 \Rightarrow \frac{\partial s_3}{\partial x} += 1 \cdot 2x|_{x=3} \end{aligned} \tag{1.11}$$

Finally we get:

$$\frac{\partial s_3}{\partial x} = 7 \tag{1.12}$$

1.5 Jacobian computation

When doing the forward and backward mode, we compute the Jacobian matrix of the function. Using this Jacobian we can do the forward mode like this:

$$J_f(x) \cdot v \qquad \text{(JVP)} \tag{1.13}$$

where v is a vector of size n (number of input variables) and the backward mode like this:

$$v^T J_f(x) \qquad \text{(VJP)} \tag{1.14}$$

Consider a function $f : \mathbb{R}^n \to \mathbb{R}^m$ then computing the full Jacobian requires n forward passes (JVP) or m backward passes (VJP). Therefore,

- If $n \ll m$, we use the forward mode because it's faster
- If $n \gg m$, we use the backward mode because it's faster
- If $n \approx m$, we can use either mode

1.6 Memory usage

The forward mode only needs to store the current value and the current derivative, so the memory usage is relatively constant.

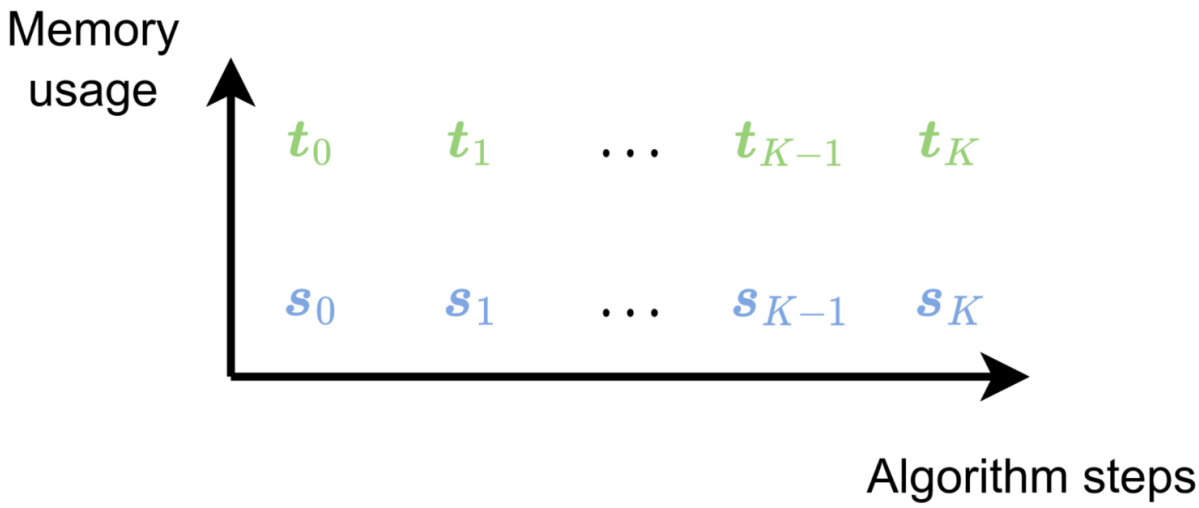


Figure 1.3: Forward mode memory usage

However, the backward mode needs to store all the intermediate values to compute the derivatives in the backward pass so the memory usage will first increase then reduce when we will start to use the derivatives previously computed.

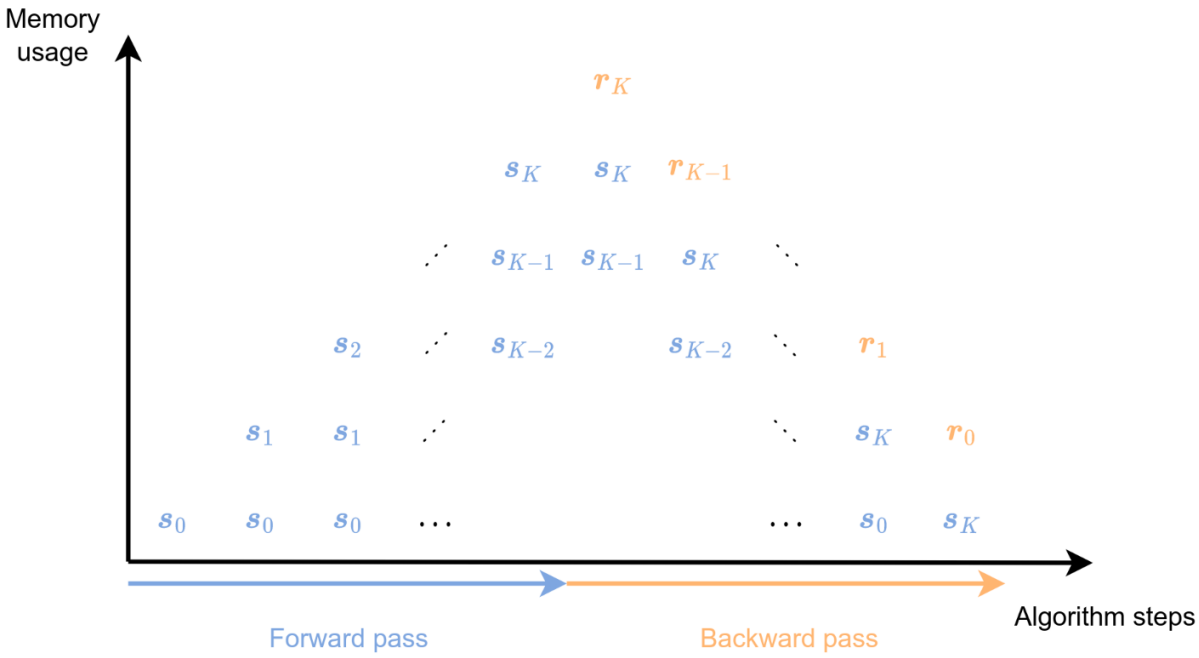


Figure 1.4: Backward mode memory usage

So the forward mode is more memory efficient than the backward mode. However, this factor may be less significant than the number of operations performed (JVP and VJP).

TODO link tangent with neural networks