

Automatic Differentiation

Benoît Legat

☐ Full Width Mode ☐ Present Mode

☰ **Table of Contents**

Differentiation approaches

Chain rule

Forward Differentiation

Reverse differentiation

Comparison

Discontinuity

Neural network

Wine example 🍷

Second-order

Differentiation approaches

We can compute partial derivatives in different ways:

1. **Symbolically**, by fixing one of the variables and differentiating with respect to the others, either manually or using a computer.
2. **Numerically**, using the formula $f'(x) \approx (f(x + h) - f(x))/h$.
3. **Algorithmically**, either forward or reverse : this is what we will explore here.

Chain rule \Leftrightarrow

Consider $f(x) = f_3(f_2(f_1(x)))$. If we don't have the expression of f_1 but we can only evaluate $f_i(x)$ or $f'(x)$ for a given x ? The chain rule gives

$$f'(x) = f'_3(f_2(f_1(x))) \cdot f'_2(f_1(x)) \cdot f'_1(x).$$

Let's define $s_0 = x$ and $s_k = f_k(s_{k-1})$, we now have:

$$f'(x) = f'_3(s_2) \cdot f'_2(s_1) \cdot f'_1(s_0).$$

Two choices here:

Forward	Reverse
$t_0 = 1$	$r_3 = 1$
$t_k = f'_k(s_{k-1}) \cdot t_{k-1}$	$r_k = r_{k+1} \cdot f'_{k+1}(s_k)$

Forward Differentiation ⇔

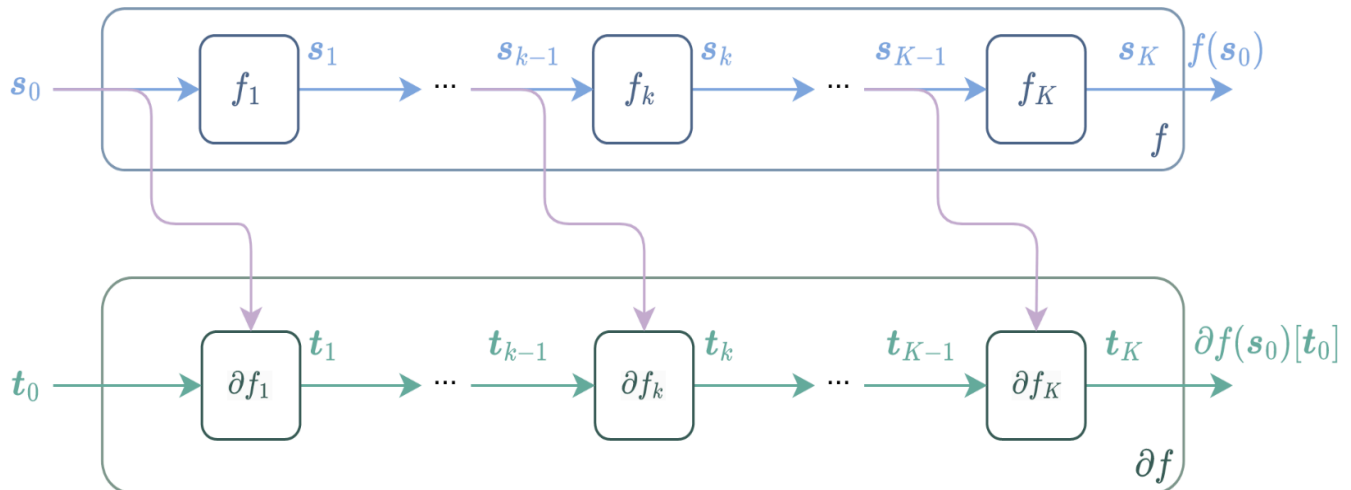


Figure 8.1

Implementation ⇔

```
1 struct Dual{T}
2     value::T # s_k
3     derivative::T # t_k
4 end
```

```
1 Base.:- (x::Dual{T}) where {T} = Dual(-x.value, -x.derivative)
```

```
1 Base.:*(x::Dual{T}, y::Dual{T}) where {T} = Dual(x.value * y.value, x.value *
y.derivative + x.derivative * y.value)
```

```
► Dual(-3, -10)
```

```
1 -Dual(1, 2) * Dual(3, 4)
```

```
f_1 (generic function with 1 method)
```

```
1 f_1(x, y) = x * y
```

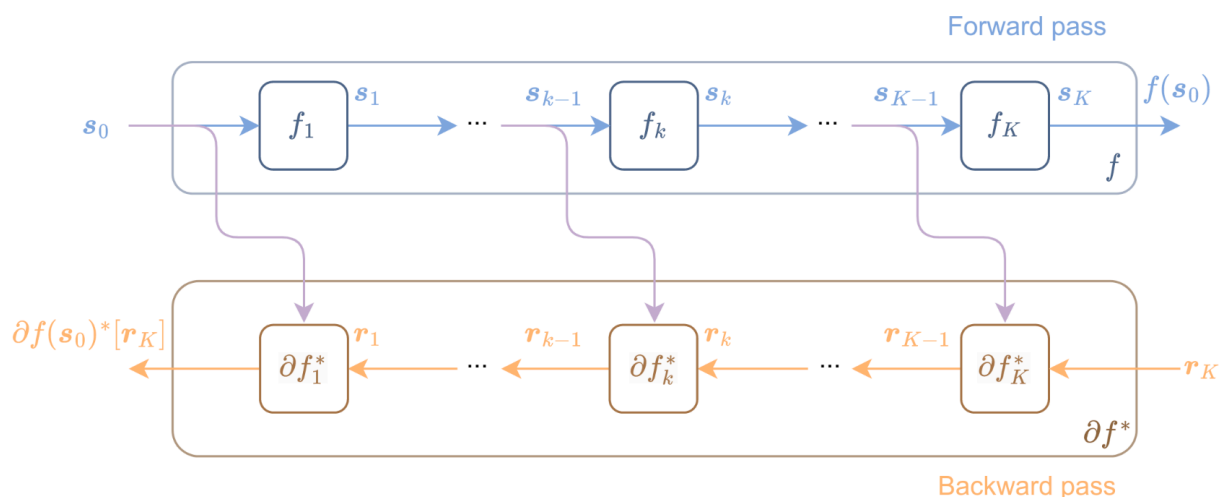
```
f_2 (generic function with 1 method)
```

```
1 f_2(s1) = -s1
```

► Dual(-3, -10)

```
1 (f_2 ◦ f_1)(Dual(1, 2), Dual(3, 4))
```

Reverse differentiation ⇔



Two different takes on the multivariate chain rule ⇔

The chain rule gives us

$$\frac{\partial f_3}{\partial x}(f_1(x), f_2(x)) = \partial_1 f_3(s_1, s_2) \cdot \frac{\partial s_1}{\partial x} + \partial_2 f_3(s_1, s_2) \cdot \frac{\partial s_2}{\partial x}$$

To compute this expression, we need the values of $s_1(x)$ and $s_2(x)$ as well as the derivatives $\partial s_1 / \partial x$ and $\partial s_2 / \partial x$.

Common to forward and reverse: Given s_1, s_2 , computes **local** derivatives $\partial_1 f_3(s_1, s_2)$ and $\partial_2 f_3(s_1, s_2)$, shortened $\partial_1 f_3, \partial_2 f_3$ for conciseness.

Forward ⇔

$$\begin{aligned} t_3 &= \partial_1 f_3 \cdot t_1 + \partial_2 f_3 \cdot t_2 \\ &= [\partial_1 f_3 \quad \partial_2 f_3] \cdot \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \\ &= \partial f_3 \cdot \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \end{aligned}$$

Reverse ⇔

$$\begin{aligned} [r_1 \quad r_2] &+= r_3 \cdot \partial f_3 \\ &+= r_3 \cdot [\partial_1 f_3 \quad \partial_2 f_3] \\ &+= [r_3 \cdot \partial_1 f_3 \quad r_3 \cdot \partial_2 f_3] \end{aligned}$$

Reverse* ⇔

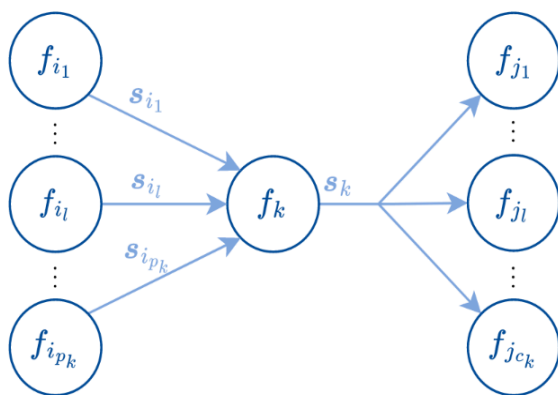
$$\begin{aligned} \begin{bmatrix} r_1 \\ r_2 \end{bmatrix} &+= \partial f_3^* \cdot r_3 \\ &+= \begin{bmatrix} \partial_1 f_3 \\ \partial_2 f_3 \end{bmatrix} \cdot r_3 \\ &+= \begin{bmatrix} \partial_1 f_3 \cdot r_3 \\ \partial_2 f_3 \cdot r_3 \end{bmatrix} \end{aligned}$$

When using automatic differentiation, don't forget that we must always evaluate the derivatives. For the following example we choose to evaluate it in $x = 3$

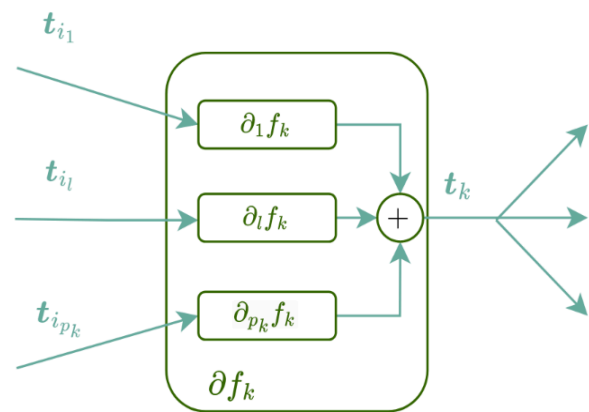
► Apply the automatic differentiation to $s_3 = f_3(s_1, s_2) = s_1 + s_2$, with $s_1 = f_1(x) = x$ and $s_2 = f_2(x) = x^2$

Forward tangents ⇔

Forward pass

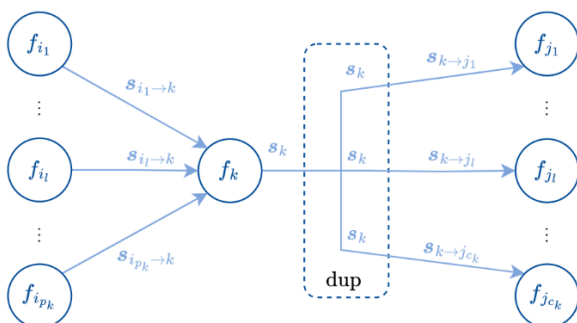


Forward mode

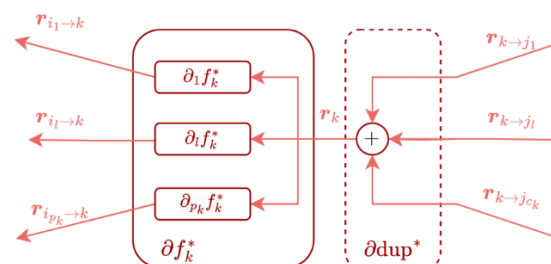


Reverse tangents ⇔

Forward pass

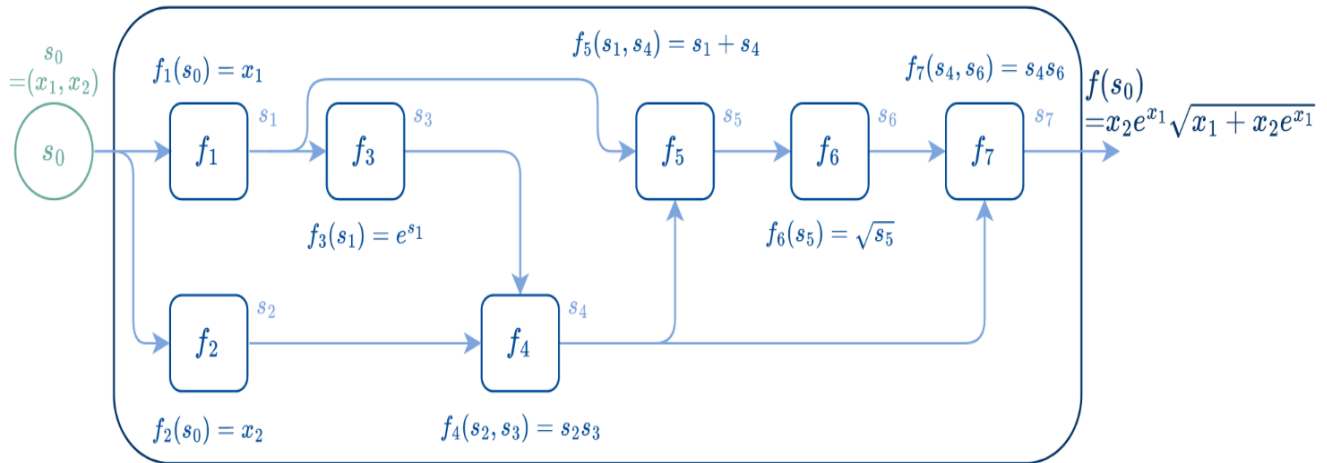


Reverse mode



► Why is ∂dup^* a sum ?

Expression graph ⇔



► Can this directed graph have cycles ?

► What happens if f_4 is handled before f_5 in the backward pass ?

► How to prevent this from happening ?

Comparison ⇔

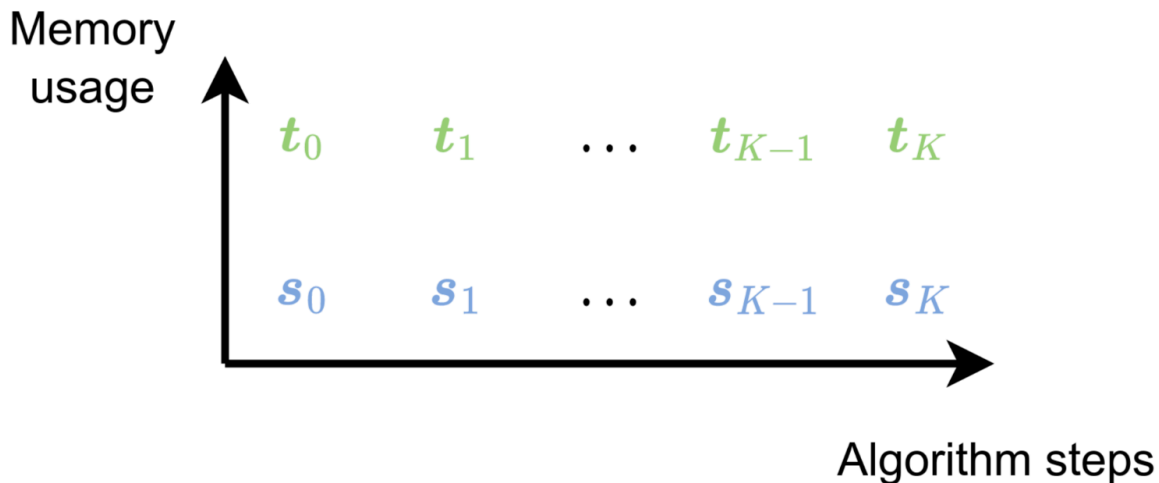
- Forward mode of $f(x)$ with dual numbers $\text{Dual.}(x, v)$ computes Jacobian-Vector Product (JVP) $J_f(x) \cdot v$
- Reverse mode of $f(x)$ computes Vector-Jacobian Product (VJP) $v^\top J_f(x)$ or in other words $J_v(x)^\top v$

► How can we compute the full Jacobian ?

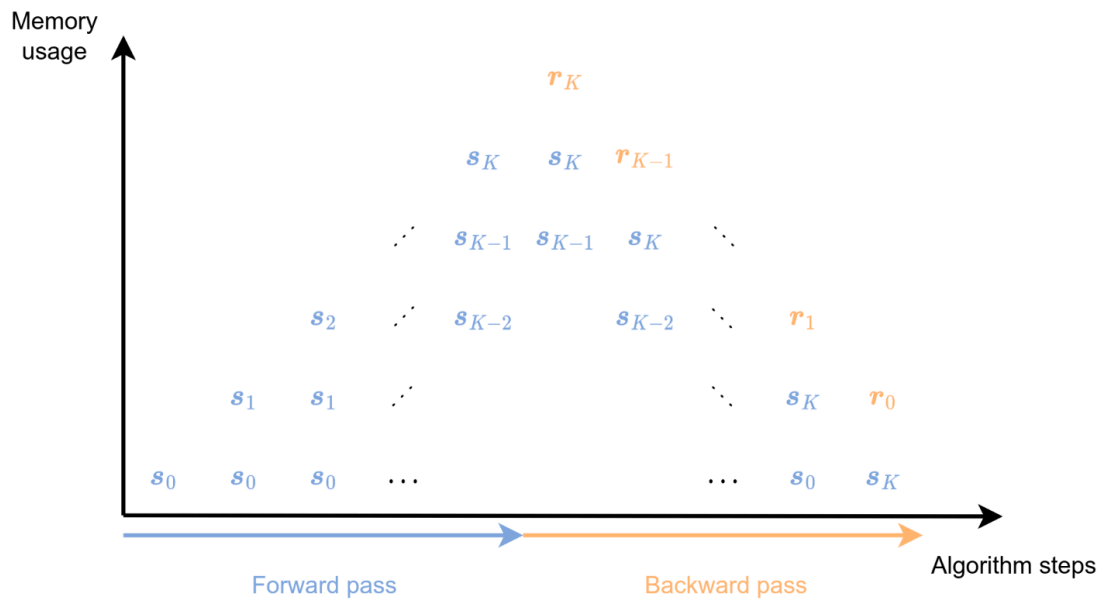
► When is each mode faster than the other one to compute the full Jacobian ?

► When is the speed of numerical differentiation comparable to autodiff ?

Memory usage of forward mode ⇔

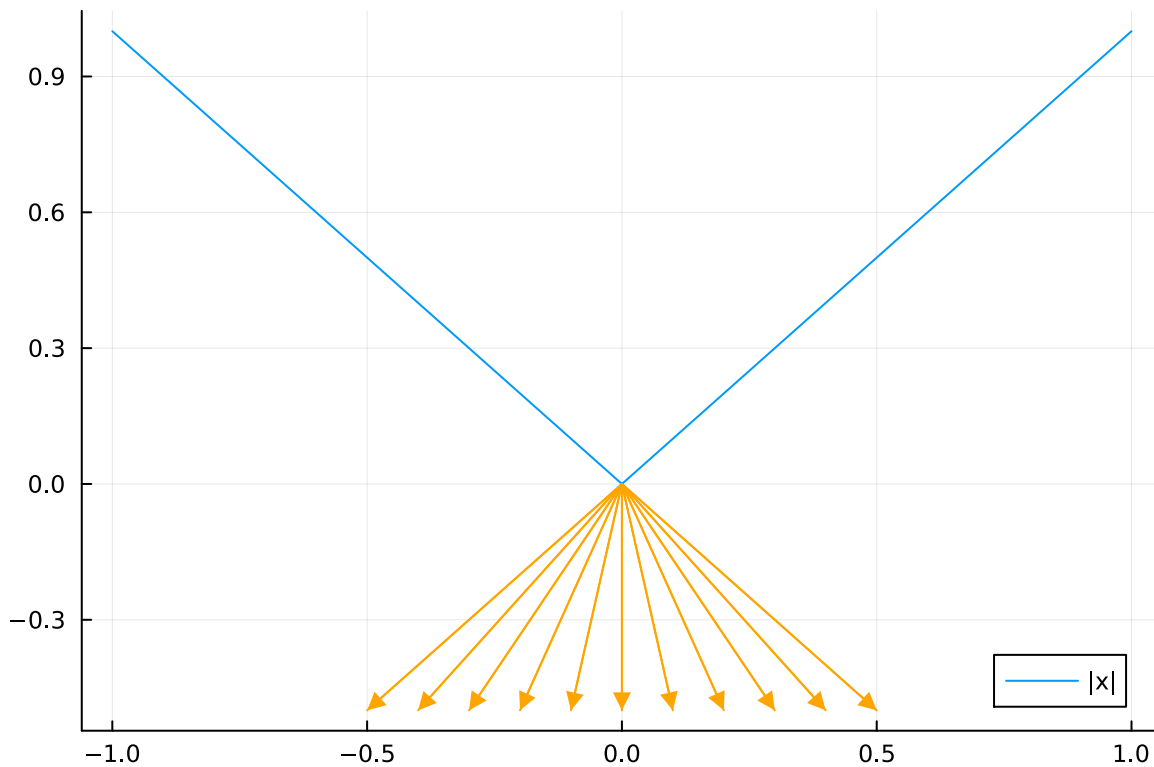


Memory usage of reverse mode \Leftrightarrow



Discontinuity ⇔

► Is the function $|x|$ is differentiable at $x = 0$?



► What about returning a convex combination of the derivative from the left and right ?

Forward mode ⇔

```
abs (generic function with 1 method)
```

```
1 abs(x) = ifelse(x < 0, -x, x)
```

```
abs_bis (generic function with 1 method)
```

```
1 abs_bis(x) = ifelse(x > 0, x, -x)
```

```
1 Base.isless(x::Dual, y::Real) = isless(x.value, y)
```

```
1 Base.isless(x::Real, y::Dual) = isless(x, y.value)
```

► Dual(0, 1)

```
1 abs(Dual(0, 1))
```

► Dual(0, -1)

```
1 abs_bis(Dual(0, 1))
```

Neural network ⇔

Two equivalent approaches, b_k is a **column** vector, S_i, X, W_i, Y are matrices.

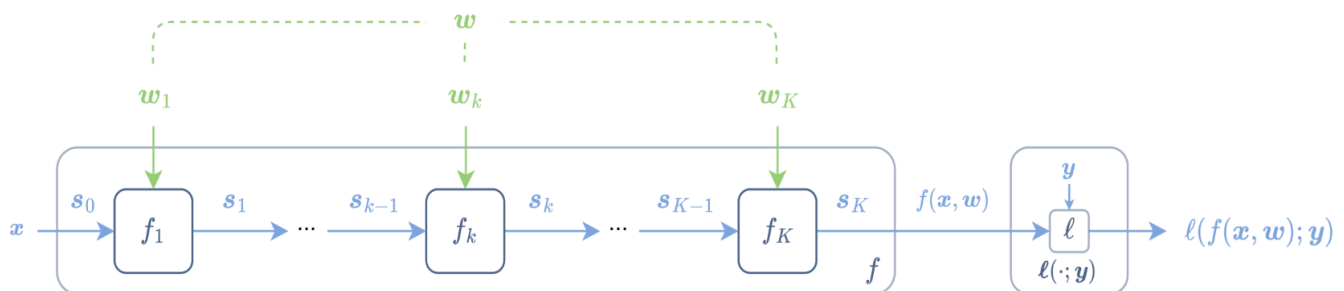
Right-to-left ⇔

$$\begin{aligned} S_0 &= X \\ S_{2k-1} &= W_k S_{2k-2} + b_k \mathbf{1}^\top \\ S_{2k} &= \sigma(S_{2k-1}) \\ S_{2H+1} &= W_{k+1} S_{2H} \\ S_{2H+2} &= \ell(S_{2H+1}; Y) \end{aligned}$$

Left-to-right ⇔

$$\begin{aligned} S_0 &= X \\ S_{2k-1} &= S_{2k-2} W_k + \mathbf{1} b_k^\top \\ S_{2k} &= \sigma(S_{2k-1}) \\ S_{2H+1} &= S_{2H} W_{k+1} \\ S_{2H+2} &= \ell(S_{2H+1}; Y) \end{aligned}$$

Evaluation ⇔



Matrix multiplication (Vectorized way) ⇔

Useful: $\text{vec}(AXB) = (B^\top \otimes A)\text{vec}(X)$

$$\begin{aligned} F(X) &= AX \\ G(\text{vec}(X)) &\triangleq \text{vec}(F(X)) = (I \otimes A)\text{vec}(X) \\ J_G &= (I \otimes A) \\ J_G^\top \text{vec}(R) &= (I \otimes A^\top)\text{vec}(R) \\ \partial F^*[R] &= \text{mat}(J_G^\top \text{vec}(R)) = A^\top R \end{aligned}$$

► How should we store the Jacobian in the forward pass to save it for the backward pass ?

Matrix multiplication (Scalar product way) ⇔

The adjoint of a linear map A for a given scalar product $\langle \cdot, \cdot \rangle$ is the linear map A^* such that

$$\forall x, y, \quad \langle A(x), y \rangle = \langle x, A^*(y) \rangle.$$

For the scalar product

$$\langle X, Y \rangle = \sum_{i,j} X_{ij} Y_{ij} = \langle \text{vec}(X), \text{vec}(Y) \rangle = \text{tr}(XY^\top), \quad A^* = A^\top$$

Now, given a forward tangent T and a reverse tangent R

$$\langle AT, R \rangle = \langle T, A^\top R \rangle$$

so the backward pass computes $A^\top R$.

► How to prove that $A^* = A^\top$?

Broadcasting (Vectorized way) ⇔

Consider applying a scalar function f (e.g. **tanh**) to each entry of a matrix X .)

$$\begin{aligned} (F(X))_{ij} &= f(X_{ij}) = f.(X) \\ G(\text{vec}(X)) &\triangleq \text{vec}(F(X)) = \text{vec}(f.(X)) \\ J_G &= \text{Diag}(\text{vec}(f'.(X))) \\ J_G \text{vec}(T) &= \text{Diag}(\text{vec}(f'.(X))) \text{vec}(T) \\ \partial F[T] &= \text{mat}(J_G \text{vec}(T)) = f'.(X) \odot T \\ J_G^\top \text{vec}(R) &= \text{Diag}(\text{vec}(f'.(X))) \text{vec}(R) \\ \partial F^*[R] &= \text{mat}(J_G^\top \text{vec}(R)) = f'.(X) \odot R \end{aligned}$$

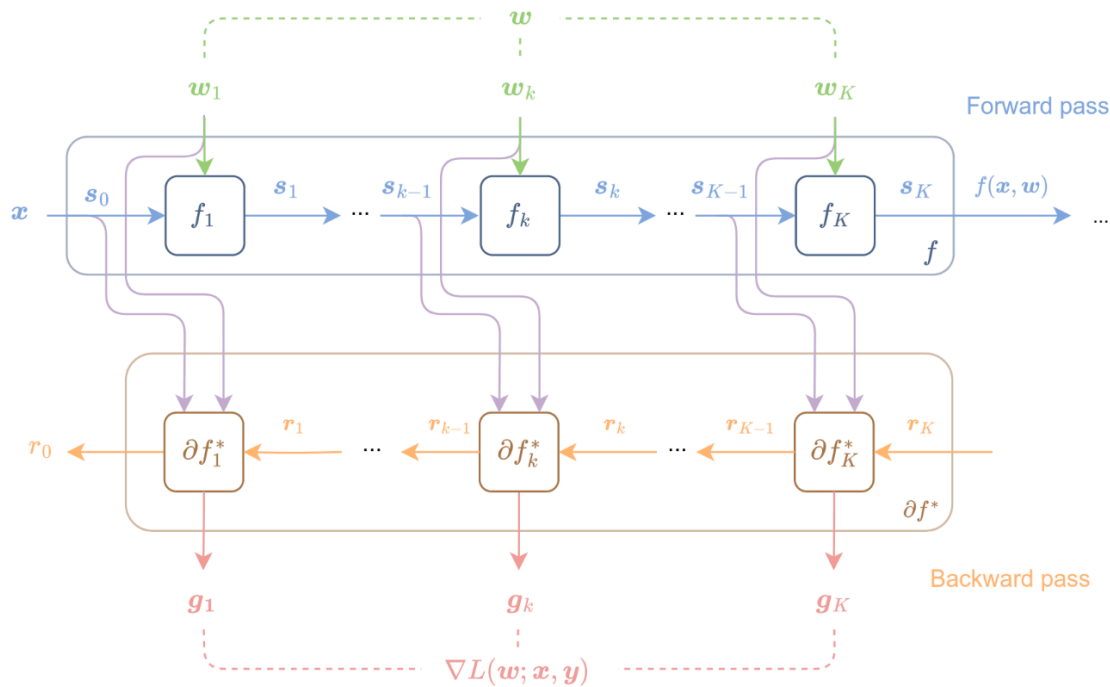
Broadcasting (Scalar product way) ⇔

$$\langle f'.(X) \odot T, R \rangle = \langle T, f'.(X) \odot R \rangle.$$

► Let $A(X) = B \odot X$, what is the adjoint A^* ?

► What should be saved for the backward pass ?

Putting everything together ⇔



Product of Jacobians ⇔

Suppose that we need to differentiate a composition of functions: $(f_n \circ f_{n-1} \circ \dots \circ f_2 \circ f_1)(w)$. For each function, we can compute a jacobian given the value of its input. So, during a forward pass, we can compute all jacobians. We now just need to take the product of these jacobians:

$$J_n J_{n-1} \dots J_2 J_1$$

While the product of matrices is associative, its computational complexity depends on the order of the multiplications! Let $d_i \times d_{i-1}$ be the dimension of J_i .

► What is the complexity of forward mode

► What is the complexity of reverse mode

► What about the complexity of meeting in the middle between k and $k + 1$?

► Which mode should be used depending on the d_i ?

► What about neural networks ?

Wine example

```
wine = dataset Wine:
  metadata => Dict{String, Any} with 4 entries
  features  => 13x178 Matrix{Float64}
  targets   => 1x178 Matrix{Int64}
  dataframe => nothing
```

```
1 wine = MLDatasets.Wine(; as_df = false)
```

normalise (generic function with 1 method)

```
1 function normalise(x)
2    $\mu$  = Statistics.mean(x, dims=2)
3    $\sigma$  = Statistics.std(x, dims=2, mean= $\mu$ )
4   return (x .-  $\mu$ ) ./  $\sigma$ 
5 end
```

```
X = 13x178 Matrix{Float32}:
 1.51434  0.245597  0.196325  1.68679  ...  0.331822  0.208643  1.39116
-0.560668 -0.498009  0.0211715 -0.345835  1.73984  0.227053  1.57871
 0.2314   -0.825667  1.10621   0.486554  -0.38826  0.0126963  1.36137
-1.1663   -2.48384  -0.267982 -0.806975  0.151234  0.151234  1.49872
 1.90852   0.018094  0.0881098  0.9283    1.41841  1.41841  -0.261969
 0.806722  0.567048  0.806722  2.48444  ... -1.12665  -1.03078  -0.391646
 1.03191   0.731565  1.21211   1.4624    -1.3408  -1.35081  -1.27072
-0.657708 -0.818411  -0.497005 -0.979113  0.547563  1.35108  1.59213
 1.22144  -0.543189  2.12996   1.02925  -0.420888 -0.228701 -0.420888
 0.251009 -0.292496  0.268263  1.18273   2.21798  1.82976  1.78663
 0.361158  0.404908  0.317409 -0.426341  ... -1.60759  -1.56384  -1.52009
 1.84272   1.11032  0.786369  1.18074  -1.48127  -1.39676  -1.42493
 1.01016   0.962526  1.39122   2.32801   0.279786  0.295664  -0.593486
```

```
1 X = Float32.(normalise(wine.features))
```

```
y =
1x178 Matrix{Float32}:
-1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 ... 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
```

```
1 y = Float32.(wine.targets .- 2)
```

Neural network

h =  16

```
W =
▶ [16×13 Matrix{Float32}:
 0.941756  0.579923  0.778487  0.267075  ...  0.876697  0.0158728  0.0235253  0.9:
1 W = [rand(Float32, h, size(X, 1)), rand(Float32, size(y, 1), h)]
```

```
55.4048f0
```

```
1 sum((W[2] * tanh.(W[1] * X) - y).^2) / size(y, 2)
```

Forward mode ⇄

forward_pass (generic function with 1 method)

```
1 function forward_pass(W, X, y)
2     W1, W2 = W
3     y_1 = tanh.(W1 * X)
4     local_der_tanh = 1 .- y_1.^2
5     local_der_mse = 2 * (W2 * y_1 - y) / size(y, 2)
6     return local_der_tanh, local_der_mse
7 end
```

forward_diff (generic function with 1 method)

```
1 function forward_diff(W, X, y, j, k)
2     W1, W2 = W
3     T_1 = onehot(j, axes(W1, 1)) * onehot(k, axes(W1, 2))'
4     J_1, J_2 = forward_pass(W, X, y)
5     only((W2 * (J_1 .* (T_1 * X))) * J_2') # only: 1x1 matrix -> scalar
6 end
```

forward_diff (generic function with 2 methods)

```
1 function forward_diff(W, X, y)
2     [forward_diff(W, X, y, i, j) for i in axes(W[1], 1), j in axes(W[1], 2)]
3 end
```

16x13 Matrix{Float32}:

```
0.766109 -0.156714 0.153898 -0.606538 ... 0.630994 1.78044 1.21098
2.76374 -0.447135 0.44208 -2.28234 1.49692 2.48256 2.93872
0.194518 -0.118595 0.00553678 -0.239882 0.354018 0.213944 0.32721
0.433517 -0.214309 0.196413 -0.296627 0.445041 0.72567 0.438316
0.282579 0.333575 0.136095 0.211333 -0.154399 0.173398 0.0936259
0.0751439 0.438566 0.235787 0.209088 ... 0.0897279 0.112657 0.0480453
0.40879 0.286055 0.360139 -0.0517686 -0.132461 -0.00428092 0.256669
⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮
1.25234 -0.582268 0.594282 -0.56713 ... 0.834521 0.966969 1.4506
1.64608 1.09168 0.903536 -0.467423 -0.544111 0.0171324 1.05362
0.776219 1.21391 1.04492 0.29169 -0.878149 -0.474735 0.764322
0.815497 0.477986 1.13609 0.144571 -0.0679515 -0.490821 0.820148
0.453604 -0.324761 0.172441 -0.476026 0.535761 0.729985 0.378242
0.0726003 -0.258423 -0.0532104 -0.337818 ... 0.741748 0.693152 0.318222
```

```
1 if h < 200 # Forward Diff start being too slow for 'h > 200'
2   @time forward_diff(W, X, y)
3 end
```



0.014414 seconds (6.24 k allocations: 12.209 MiB)



Reverse mode ⇄

► How to deduce the backward pass for reverse mode from the forward mode ?

reverse_diff (generic function with 1 method)

```
1 function reverse_diff(W, X, y)
2   J_1, J_2 = forward_pass(W, X, y)
3   (J_1 .* (W[2]' * J_2)) * X'
4 end
```

```
16x13 Matrix{Float32}:
 0.76611   -0.156713  0.153898   -0.606538  ...  0.630994   1.78044   1.21098
 2.76374   -0.447135  0.44208    -2.28234   ...  1.49692   2.48256   2.93872
 0.194518  -0.118595  0.00553679 -0.239882   ...  0.354019  0.213944  0.32721
 0.433517  -0.214309  0.196413   -0.296627   ...  0.445041  0.72567   0.438316
 0.282579  0.333575  0.136095   0.211333   ... -0.154399  0.173398  0.0936259
 0.0751439 0.438566  0.235787   0.209088   ...  0.0897279 0.112657  0.0480453
 0.40879   0.286055  0.360139   -0.0517686  ... -0.132461 -0.00428092 0.256669
 ⋮
 1.25234   -0.582268  0.594282   -0.56713   ...  0.834521  0.966969  1.4506
 1.64608   1.09168  0.903536   -0.467423  ... -0.544111 0.0171324 1.05362
 0.776219  1.21391  1.04492    0.29169    ... -0.878149 -0.474735 0.764322
 0.815497  0.477986  1.13609    0.144571   ... -0.0679515 -0.490821 0.820148
 0.453604  -0.324761  0.172441   -0.476026  ...  0.535761  0.729985 0.378242
 0.0726002 -0.258423 -0.0532103 -0.337818  ...  0.741748  0.693152 0.318222
```

```
1 @time reverse_diff(W, X, y)
```

```
0.000063 seconds (25 allocations: 60.039 KiB)
```

GPU acceleration

```
1 if CUDA.functional()
2     X_gpu = CUDA.CuArray(X)
3     y_gpu = CUDA.CuArray(y)
4     W_gpu = CUDA.CuArray(W)
5     @time reverse_diff(W_gpu, X_gpu, y_gpu)
6 end
```

► Why is the GPU version slower than the CPU version ?

Second-order ⇔

Consider a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, we want to compute the Hessian $\nabla^2 f(x)$, defined by

$$(\nabla^2 f(x))_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

Application: Given the optimization problem:

$$\begin{aligned} \min_x f(x) \\ g_i(x) = 0 \quad \forall i \in \{1, \dots, m\} \end{aligned}$$

The Hessian of the Lagrangian $\mathcal{L}(x, \lambda) = f(x) - \lambda_1 g_1(x) - \dots - \lambda_m g_m(x)$ is obtained as

$$\nabla_x^2 \mathcal{L}(x, \lambda) = \nabla^2 f(x) - \sum_{i=1}^m \lambda_i \nabla^2 g_i(x)$$

Second-order AD ⇔

► How can the Hessian of f be computed given an AD for Jacobian and gradient.

► Does the AD need to be the same for the gradient and the Jacobian ?

Notation ⇔

- Let $f_k : \mathbb{R}^{d_{k-1}} \rightarrow \mathbb{R}^{d_k}$. $\partial f_k \triangleq \partial f_k(s_{k-1}) \in \mathbb{R}^{d_k \times d_{k-1}}$, $\partial^2 f_k \triangleq \partial^2 f_k(s_{k-1}) \in \mathbb{R}^{d_k \times d_{k-1} \times d_{k-1}}$ is a 3D array/tensor.
- Given $v \in \mathbb{R}^{d_{k-1}}$, by the product $(\partial^2 f_k \cdot v) \in \mathbb{R}^{d_k \times d_{k-1}}$, we denote the contraction of the 3rd (or 2nd since the tensor is symmetric over its last 2 dimensions) dimension:

$$(\partial^2 f_k \cdot v)_{ij} = \sum_{l=1}^{d_{k-1}} (\partial^2 f_k)_{ijl} \cdot v_l$$

- Given $\mathbf{u} \in \mathbb{R}^{d_k}$, by the product $(\mathbf{u} \cdot \partial^2 f_k) \in \mathbb{R}^{d_{k-1} \times d_{k-1}}$, we denote the contraction of the 1st dimension.

$$(\mathbf{u} \cdot \partial^2 f_k)_{ij} = \sum_{l=1}^{d_k} u_l \cdot (\partial^2 f_k)_{lij}$$

- Both $\partial^2 f_k \cdot \mathbf{v}$ and $\mathbf{u} \cdot \partial^2 f_k$ are matrices so then we're back to matrix notations.

Chain rule \Leftrightarrow

$$\begin{aligned} \frac{\partial^2(f_2 \circ f_1)}{\partial x_i \partial x_j} &= \frac{\partial}{\partial x_j} \left(\frac{\partial(f_2 \circ f_1)}{\partial x_i} \right) \\ &= \frac{\partial}{\partial x_j} \left(\partial f_2 \cdot \frac{\partial f_1}{\partial x_i} \right) \\ &= \left(\partial^2 f_2 \cdot \frac{\partial f_1}{\partial x_j} \right) \cdot \frac{\partial f_1}{\partial x_i} + \partial f_2 \cdot \frac{\partial^2 f_1}{\partial x_i \partial x_j} \end{aligned}$$

In terms of the matrices $\mathbf{J}_k = \partial f_k$ and $\mathbf{H}_{kj} = \frac{\partial}{\partial x_j} \mathbf{J}_k = \partial^2 f_k \cdot \frac{\partial s_{k-1}}{\partial x_j}$, it becomes

$$\frac{\partial^2(f_2 \circ f_1)}{\partial x_i \partial x_j} = \mathbf{H}_{2j} \cdot \frac{\partial f_1}{\partial x_i} + \mathbf{J}_2 \cdot \frac{\partial^2 f_1}{\partial x_i \partial x_j}$$

Forward on forward \Leftrightarrow

Given $\mathbf{Dual}(s_1, t_1)$ with $s_1 = \mathbf{Dual}(f_1(x), \frac{\partial f_1}{\partial x_j})$ and $t_1 = \mathbf{Dual}(\frac{\partial f_1}{\partial x_i}, \frac{\partial^2 f_1}{\partial x_i \partial x_j})$

1. Compute $s_2 = f_2(s_1) = (f_2(f_1(x)), \mathbf{J}_2 \cdot \frac{\partial f_1}{\partial x_j}) = ((f_2 \circ f_1)(x), \partial(f_2 \circ f_1)/\partial x_j)$
2. Compute $\mathbf{J}_{f_2}(s_1)$ which gives $\mathbf{Dual}(\mathbf{J}_2, \mathbf{H}_{2j})$
3. Compute

$$\begin{aligned} \mathbf{J}_{f_2}(s_1) \cdot t_1 &= \mathbf{Dual}(\mathbf{J}_2, \mathbf{H}_{2j}) \cdot \mathbf{Dual}(\frac{\partial f_1}{\partial x_i}, \frac{\partial^2 f_1}{\partial x_i \partial x_j}) \\ &= \mathbf{Dual}(\mathbf{J}_2 \cdot \frac{\partial f_1}{\partial x_i}, \mathbf{J}_2 \cdot \frac{\partial^2 f_1}{\partial x_i \partial x_j} + \mathbf{H}_{2j} \cdot \frac{\partial f_1}{\partial x_i}) \\ &= \mathbf{Dual}(\frac{\partial(f_2 \circ f_1)}{\partial x_i}, \frac{\partial^2(f_2 \circ f_1)}{\partial x_i \partial x_j}) \end{aligned}$$

► What is the closed form expression for t_k in terms of the matrices J_k and H_{kj} ?

Forward on reverse ⇔

Forward pass: Given $s_1 = \text{Dual}(f_1(x), \frac{\partial f_1}{\partial x_j})$

1. Compute $s_2 = f_2(s_1) \rightarrow$ same as forward on forward
2. Compute $J_{f_2}(s_1) \rightarrow$ same as forward on forward

Reverse pass: Given $r_2 = \text{Dual}((r_2)_1, (r_2)_2)$, compute

$$\begin{aligned} r_2 \cdot J_2 &= \text{Dual}((r_2)_1, (r_2)_2) \cdot \text{Dual}(J_2, H_{2j}) \cdot \\ &= \text{Dual}((r_2)_1 \cdot J_2, (r_2)_2 \cdot J_2 + (r_2)_1 \cdot H_{2j}) \end{aligned}$$

► Which value of r_k is solution for this recurrence equation ?

► What is the closed form expression for r_k in terms of the matrices J_k and H_{kj} ?

Reverse on forward ⇔

Forward pass: Given $s_1 = \text{Dual}(f_1(x), \frac{\partial f_1}{\partial x_i})$

1. Forward mode computes $s_2 = f_2(s_1) = (f_2(f_1(x)), J_2 \cdot \frac{\partial f_1}{\partial x_i}) = ((f_2 \circ f_1)(x), \partial(f_2 \circ f_1)/\partial x_i)$
2. The reverse mode computes the local Jacobian of this operation : $\partial s_2 / \partial s_1$. The local Jacobian of $(s_1)_1 \mapsto f_2((s_1)_1)$ is J_2 . The local Jacobian of $s_1 \mapsto \partial f_2((s_1)_1)(s_1)_2$ is $(\partial^2 f_2((s_1)_1) \cdot (s_1)_2, \partial f_2((s_1)_1)) = (\partial^2 f_2(f_1(x)) \cdot \frac{\partial f_1}{\partial x_i}, \partial f_2(f_1(x))) = (H_{2i}, J_2)$

Reverse pass:

$$\begin{aligned} (r_1)_1 &= (r_2)_1 \cdot J_2 + (r_2)_2 \cdot H_{2i} \\ (r_1)_2 &= (r_2)_2 \cdot J_2 \end{aligned}$$

► Which value of r_k is solution for this recurrence equation ?

Reverse on reverse ⇔

Forward pass (2nd):

1. Forward pass computes $s_2 = f_2(s_1) \rightarrow$ Jacobian $\partial s_2 / \partial s_1 = J_2$
2. Local Jacobian $J_2 = \partial f_2(s_1) \rightarrow$ The Jacobian is the 3D array $\partial J_2 / \partial s_1 = \partial^2 f_2$
3. Backward pass computes $r_1 = r_2 \cdot \partial f_2(s_1) \rightarrow$ Jacobian of $(s_1, r_2) \mapsto r_2 \cdot \partial f_2(s_1)$ is $(r_2 \cdot \partial^2 f_2(s_1), \partial f_2(s_1)) = (r_2 \cdot \partial^2 f_2, J_2)$. Note that here $r_2 \in \mathbb{R}^{d_k}$ is multiplying the first dimension of the tensor $\partial^2 f_2(s_1) \in \mathbb{R}^{d_k \times d_{k-1} \times d_{k-1}}$ so the result is a symmetric matrix of dimension $\mathbb{R}^{d_{k-1} \times d_{k-1}}$

Reverse pass (2nd): The result is r_0 , let \dot{r}_k be the second-order reverse tangent for r_k and \dot{s}_k be the second-order reverse tangent of s_k . We have

$$\begin{aligned}\dot{r}_2 &= J_2 \cdot \dot{r}_1 \\ \dot{s}_1 &= (r_2 \cdot \partial^2 f_2(s_1)) \cdot \dot{r}_1 + \dot{s}_2 \cdot J_2\end{aligned}$$

► Which value of \dot{s}_k, \dot{r}_k is solution for this recurrence equation ?

► What is the difference with reverse on forward and forward on reverse ?

Acknowledgements and further readings ⇔

- Dual is inspired from [ForwardDiff](#)
- Node is inspired from [micrograd](#)
- [Here](#) is a good intro to AD
- Figures are from the [The Elements of Differentiable Programming book](#)

The End

Utils

```
1 using Plots, PlutoUI, PlutoUI.ExperimentalLayout, HypertextLiteral; @html, @html_str
   PlutoTeachingTools, DataFrames, MLDatasets, Statistics, CUDA, OneHotArrays
```

img (generic function with 3 methods)

qa (generic function with 2 methods)