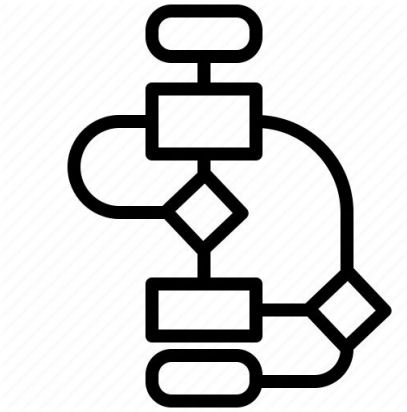


# LINMA2472 - Algorithm in data science

Simon Desmidt

Academic year 2025-2026 - Q1

UCLouvain

# Contents

# Automatic Differentiation

Automatic Differentiation (AD) 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.) By mean of the chain rule, applied to obtain the all of a function's derivatives, its gradient can be exactly and efficiently computed.

Automatic differentiation is widely used in machine learning because neural networks requires the computation of the loss function's gradient with respect to the model's parameters (weights and biases). This is necessary to update them during the training process (e.g. with a gradient descent) and it would be a pain in the ass to compute this manually for each node.

## 1.1 Chain rule

Assume $f$ is the composition of $m$ functions. The chain rule gives

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

Defining

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

gives

$$f'(x) = f'_m(s_{m-1}) \cdot f'_{m-1}(s_{m-2}) \cdot \dots \cdot f'_2(s_1) \cdot f'_1(s_0) \tag{1.3}$$

Based on this expression, 2 ways of applying the chain rule can be defined: forward and backward differentiation.

## 1.2 Forward differentiation

Also called forward mode, this algorithm consists in propagating forward the derivative and the values at the same time. Propagating the values forward is called a **forward pass**. This process can be represented by the graph on figure 1.1. The blue part represents the values and the green part shows the derivatives.

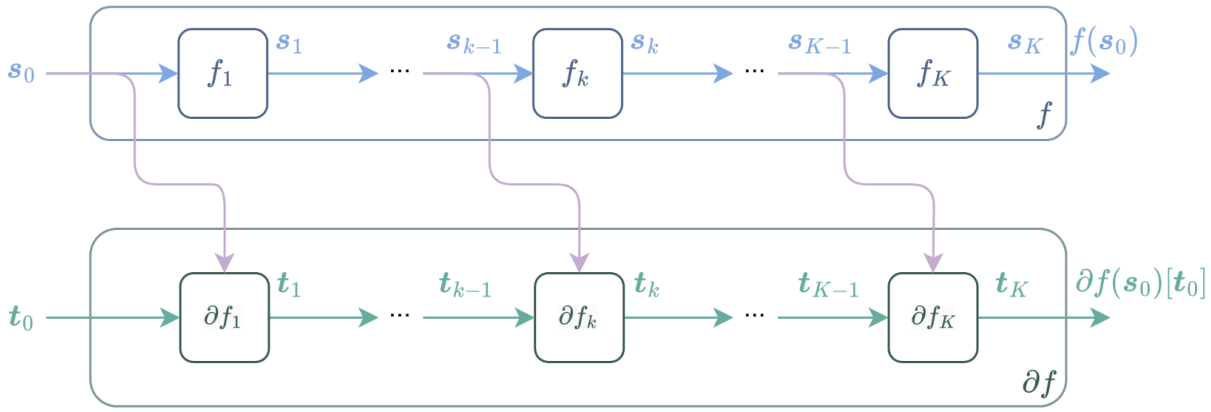



Figure 1.1: Forward differentiation

The algorithm follows:

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

This process is repeated for every input variables. Consequently, it is *very efficient* for functions with a *small number of input* variables and bad for functions with a large number of inputs.

The forward differentiation algorithm is the easiest to implement.

## 1.3 Backward differentiation

Also called backward mode, the algorithm consists in propagating the values forward and the derivatives backward *in one pass*. Propagating the derivatives backward is called a **backward pass**. This process can be represented by the graph on figure 1.2. The blue part represents the values and the orange part the derivatives.

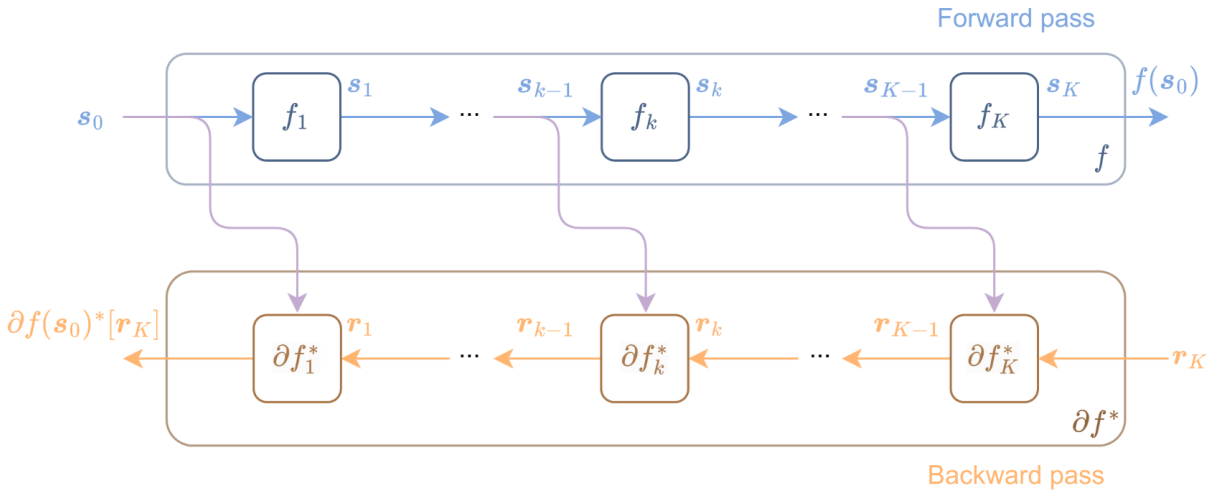



Figure 1.2: Backward differentiation

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. The algorithm follows the

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 heavier to implement but it is very efficient for functions with a large number of input variables and a small number of output variables. Typically, the backward mode is preferred for neural networks, where there is typically *only one* output variable: the loss.

## 1.4 Computational graph and multivariate differentiation

### 1.4.1 Computational graph

To represent the computation of a function, a computational graph can be used. It is a directed acyclic graph where the nodes represent the operations and the edges represent the variables. For instance, 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:

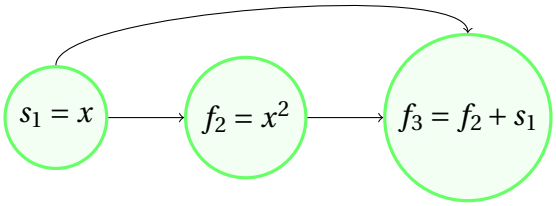



Figure 1.3: Directed Acyclic Graph (DAG) of $f_3$

### 1.4.2 Multivariate differentiation

Let's consider the function displayed on figure 1.3.

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

The chain rule gives

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

In forward mode, the values and the derivatives are propagated forward together. When a node has multiple inputs, it is necessary to use the chain rule.

Evaluating $f_3$ at $x = 3$ leads to

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

In backward mode, the gradient accumulators needs to be initialized to 0.

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

Next, the intermediate values are computed in one forward pass

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

Eventually giving

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

## 1.5 Jacobian computation

Forward and backward mode allows the indirect computation of the function's Jacobian matrix. Reminding that, for a function $f : \mathbb{R}^n \to \mathbb{R}^m$, the jacobian matrix will have dimensions $m \times n$, the following intepretations are possible.

### 1.5.1 Forward mode

Forward mode computes the function's Jacobian matrix by successive Jacobian-vector products, one for each variable.

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

For each input variable, the Jacobian-vector product, representing the directional derivative, is computed with a one-encoding vector $v$ of the said variable [1]. $v$ represents the direction of the input variable. The directional derivative may be seen as answering to

> How does a perturbation to the input propagates to the outputs?

The full jacobian matrix is obtained as the horizontal concatenation of all Jacobian-vector products computed for each input variable.

### 1.5.2 Backward mode

Backward mode computes the vector-Jacobian products.

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

Here, $v^T$ is the output covector of size $m$. It is a one-hot encoding of one of the outputs. The *VJP* answers to

[1] The one-hot encoding of a variable is a vector of length $n$ filled with zeros, with 1 set at the encoded variable's position.

> Which input perturbations matter for this specific output?

In backward mode, the full jacobian matrix is obtained by vertical concatenation of all vector-Jacobian products, computed per output.

### 1.5.3 Comparison

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

Therefore,

- If $n \ll m$, the forward mode is faster
- If $n \gg m$, the backward mode is faster
- If $n \approx m$, none prevail

## 1.6 Memory usage

The forward mode only needs to store the current value and the current derivative. The memory usage stays approximatively constant.

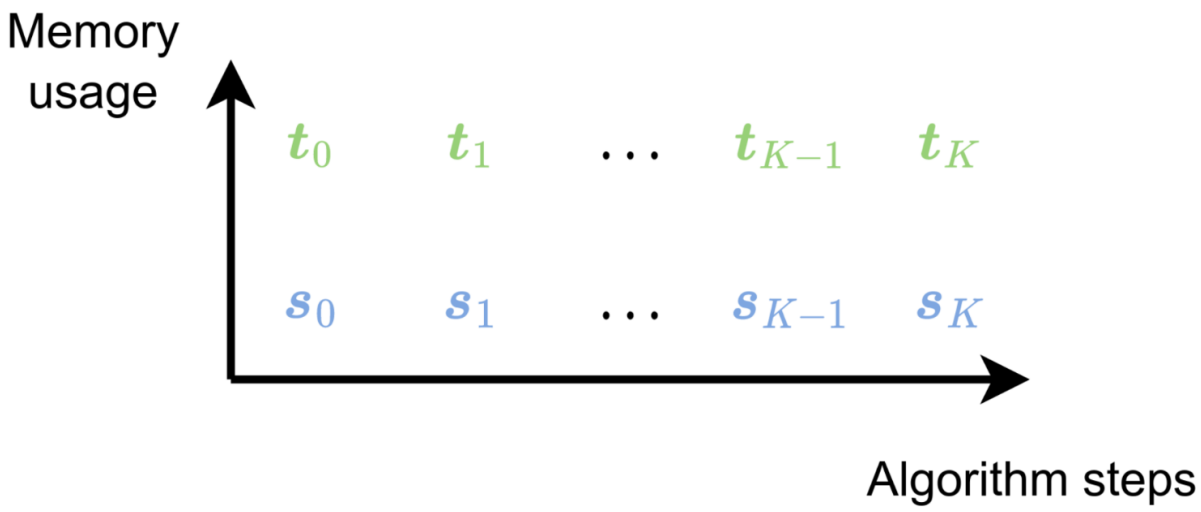



Figure 1.4: Forward mode memory usage

The backward mode, however, needs to store all intermediate values to compute the derivatives in the backward pass. As a consequence, the memory usage will increase during the forward pass, and then reduce during the derivatives computation, in the backward pass.

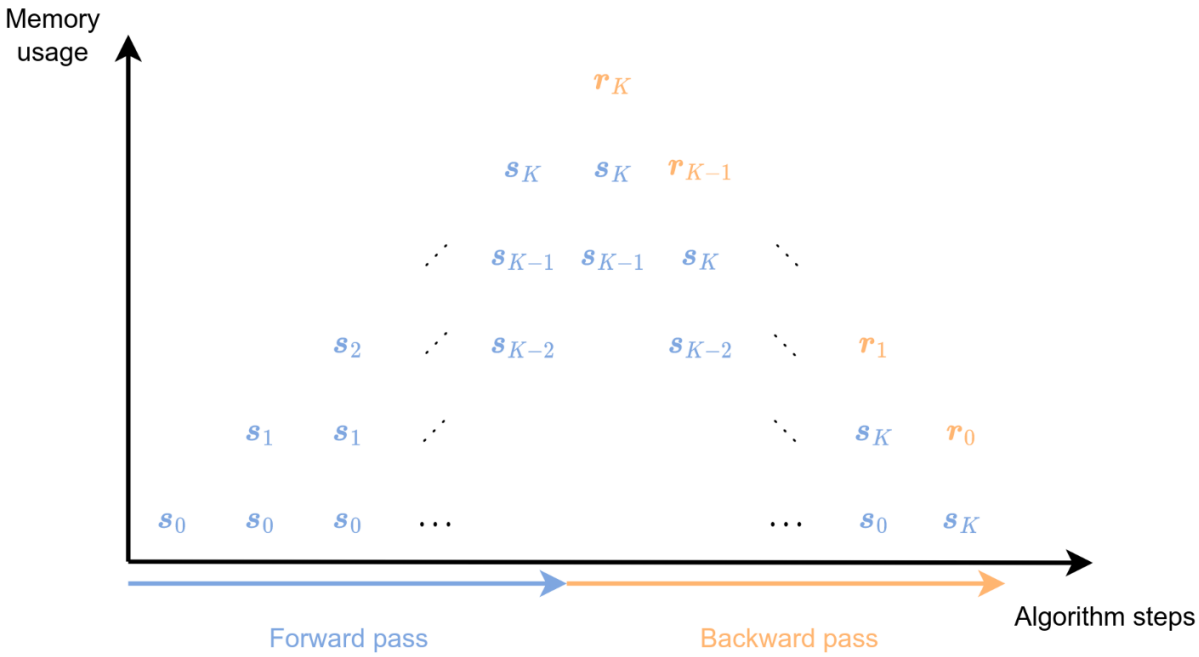



Figure 1.5: Backward mode memory usage

Concluding with this section, the forward mode is deemed more memory efficient than the backward mode. However, this aspect is generally less significant than the number of operations performed (JVP vs VJP).

## 1.7 Second order AD

Automatic Differentiation is also able to compute higher-order derivatives, such as the Hessian.

For any function $f : \mathbb{R}^n \to \mathbb{R}^m$, the Hessian, per output, is a matrix of dimensions $n \times n$. Its entries can be computed as

$$(\nabla^2 f(x))_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j} \tag{1.15}$$

The Hessian can also be seen as the Jacobian of the function's gradient

$$\nabla^2 f(x) = J_{\nabla f}(x) \tag{1.16}$$

This last definition clearly shows that AD systems can be used to obtain the Hessian, by computing the gradient and the Jacobian of $f$ as already seen. Since, the Jacobian and the gradient can be both computed independently, different AD modes (forward / backward) can be used to compute each one of them.

As a result, 4 Hessian's computation strategies can be defined [2].

[2] Since the Hessian per output is $n \times n$, the full Hessian is a 3D tensor of dimensions $m \times n \times n$. In the following, $m = 1$ will always be assumed for the sake of simplicity, but the reasoning stays the same.

### 1.7.1 Prelude

#### Rewriting the Hessian

Based on the chain rule, the Hessian's entries $\frac{\partial^2(f_2 \circ f_1)}{\partial x_i \partial x_j}$ can be rewritten into a more AD-suitable expression. Let $s_0 = x$ and $s_k = f_k(s_{k-1})$ for $k \geq 1$. Denote by $J_k \triangleq J_{f_k}(s_{k-1})$ the Jacobian of $f_k$ evaluated at $s_{k-1}$.

$$\begin{aligned} \frac{\partial^2(f_2 \circ f_1)}{\partial x_i \partial x_j} &= \frac{\partial}{\partial x_j}\left(J_2 \frac{\partial f_1}{\partial x_i}\right) \\ &= \left(\frac{\partial J_2}{\partial s_1}\frac{\partial f_1}{\partial x_j}\right)\frac{\partial f_1}{\partial x_i} + J_2 \frac{\partial^2 f_1}{\partial x_i \partial x_j} \end{aligned} \tag{1.17}$$

Introducing $\mathbf{H}_{kj} \triangleq \frac{\partial \mathbf{J}_k}{\partial x_j} = \frac{\partial \mathbf{J}_k}{s_{k-1}}\mathbf{J}_{k-1}$, the final expression reads

$$\frac{\partial^2(f_2 \circ f_1)}{\partial x_i \partial x_j} = \mathbf{H}_{2j}\frac{\partial f_1}{\partial x_i} + \mathbf{J}_2 \frac{\partial^2 f_1}{\partial x_i \partial x_j} \tag{1.18}$$

#### Dual numbers

*Dual numbers* are a concept borrowed from algebra. Initially introduced in 1873 to represent the angle between lines in space, it can also be applied to automatic differentiation.

A Dual number is an expression of the form $a + b\varepsilon$ where $a$ and $b$ are real numbers and $\varepsilon$ is a symbol such that $\varepsilon^2 = 0$ with $\varepsilon \neq 0$. This allows defining the product of two Dual numbers as

$$(a + b\varepsilon)(c + d\varepsilon) = ac + (ad + bc)\varepsilon \tag{1.19}$$

One can readily see an application equivalent to the Taylor's first-order approximation, via the Taylor series expansion.

$$f(a + b\varepsilon) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a) b^n \varepsilon^n}{n!} \overset{3}{=} f(a) + bf'(a)\varepsilon \tag{1.20}$$

Back to the automatic differentiation concept, $f(a)$ is the function's value evaluated at $a$, while $bf'(a)\varepsilon$ is the Jacobian-vector product evaluated at $a$.

- $b$ represents the tangent vector
- $f'(a)$ is the derivative (Jacobian) evaluated at $a$ in the scalar-output case

Dual numbers encode *exactly* the derivative of a function.

In the following sections, this process is extended to multivariate functions.

### 1.7.2 Forward on forward

> "Directional derivative of a directional derivative"

**Mathematically**

The forward on forward strategy computes

$$D(D(f)[v])[w] = w^T H v \tag{1.21}$$

where $v, w$ may be any arbitrary direction vectors. Usually, when computing the full Hessian, they are taken as the one-hot encodings of the variables the differentiation is made with respect to.

**Algorithmically**

Defining $\text{Dual}(s_1 \mid t_1)$, with $s_1 = \text{Dual}\left(f_1(x) \;\middle|\; \frac{\partial f_1}{\partial x_j}\right)$ and $t_1 = \text{Dual}\left(\frac{\partial f_1}{\partial x_i} \;\middle|\; \frac{\partial^2 f_1}{\partial x_i \partial x_j}\right)$, the forward-on-forward algorithm reads as follows.

1. Compute $s_2 = f_2(s_1) = \text{Dual}\left(f_2(f_1(x)) \;\middle|\; \mathbf{J}_2 \frac{\partial f_1}{\partial x_j}\right)$
2. Compute $\mathbf{J}_{f_2}(s_1)$, which gives $\text{Dual}(\mathbf{J}_2 \mid \mathbf{H}_{2j})$
3. Compute

$$\begin{aligned} t_2 &= \mathbf{J}_{f_2}(s_1)\, t_1 \\ &= \text{Dual}(\mathbf{J}_2 \mid \mathbf{H}_{2j}) \text{Dual}\left(\frac{\partial f_1}{\partial x_i} \;\middle|\; \frac{\partial^2 f_1}{\partial x_i \partial x_j}\right) \\ &= \text{Dual}\left(\mathbf{J}_2 \frac{\partial f_1}{\partial x_i} \;\middle|\; \mathbf{J}_2 \frac{\partial^2 f_1}{\partial x_i \partial x_j} + \mathbf{H}_{2j} \frac{\partial f_1}{\partial x_i}\right) \end{aligned} \tag{1.22}$$

4. Repeat

This process can be summarized as the following equations, with $g_k(x) = f_k \circ \cdots \circ f_1$.

$$\begin{cases} s_k &= \text{Dual}\left(g_k(x) \;\middle|\; \frac{\partial g_k}{\partial x_j}\right) \\ t_k &= \text{Dual}\left(\frac{\partial g_k}{\partial x_i} \;\middle|\; \frac{\partial^2 g_k}{\partial x_i \partial x_j}\right) \end{cases} \tag{1.23}$$

⚠ As one may see, the value of $w^T H v$ is a scalar. The full process must be repeated for every pair of input variables, this strategy requires $n^2$ evaluations and is therefore impractical except for very small $n$.

### 1.7.3 Forward on reverse

> "Directional derivative of the gradient"

**Mathematically**

The forward on reverse strategy computes the Hessian-Vector Product (HVP)

$$D(\nabla f)[v] = H v \tag{1.24}$$

**Algorithmically**

The forward-on-reverse strategy consists in applying forward-mode AD to the reverse-mode computation of the gradient. Operationally, this results in:

1. a forward pass propagating both primal values and tangents,
2. a backward pass in which adjoint variables are augmented with tangent components.

Defining $s_1 \triangleq \text{Dual}\left(f_1(x) \;\middle|\; \dfrac{\partial f_1}{\partial x_j}\right)$

1. Compute $s_2 = f_2(s_1)$
2. Compute $\mathbf{J}_{f_2}(s_1)$ which gives $\text{Dual}(\mathbf{J}_2, \mathbf{H}_{2j})$

The backward pass follows.

Defining $r_2 \triangleq \text{Dual}(r_{2,1} \mid r_{2,2})$, the computation is

$$\begin{aligned} r_2\mathbf{J}_2 &= \text{Dual}(r_{2,1} \mid r_{2,2})\text{Dual}(\mathbf{J}_2 \mid \mathbf{H}_{2j}) \\ &= \text{Dual}(r_{2,1}\mathbf{J}_2 \mid r_{2,1}\mathbf{H}_{2j} + r_{2,2}\mathbf{J}_2) \end{aligned} \tag{1.25}$$

Using this last equation, the recurrence relation is obtained.

$$r_k = \text{Dual}\left(\frac{\partial f}{\partial s_k} \;\middle|\; \frac{\partial^2 f}{\partial s_k \partial x_j}\right) \tag{1.26}$$

where the backward recurrence follows the same graph structure as reverse-mode AD, and each adjoint update is performed using Dual arithmetic.

Here, the tangent component of each adjoint stores the mixed second derivative.

### 1.7.4 Reverse on forward

**Mathematically**

The reverse-on-forward strategy computes the Vector-Hessian Product (VHP)

$$D(D(f)[v])^*[w] = H^T w = Hw \tag{1.27}$$

The last equation stands due to the symmetric nature of the Hessian matrix.

**Algorithmically**

The reverse-on-forward strategy starts with a forward pass.

Defining $s_1 \triangleq \text{Dual}\left(f_1(x) \;\middle|\; \dfrac{\partial f_1}{\partial x_i}\right)$ [4], the algorithm follows

1. Compute $s_2 = f_2(s_1) = \text{Dual}\left(f_2(s_1) \;\middle|\; \mathbf{J}_2 \dfrac{\partial f_1}{\partial x_i}\right)$

[4] Note the difference with the previous modes ($j \to i$)

2. The reverse mode computes the local Jacobian.

   Since $s_1$ and $s_2$ are Dual-valued variables, the Jacobian $\partial s_2/\partial s_1$ is taken with respect to the primal and tangent components.

$$\frac{\partial s_2}{\partial s_1} = \begin{bmatrix} \mathbf{J}_2 & 0 \\ \mathbf{H}_{2i} & \mathbf{J}_2 \end{bmatrix} \tag{1.28}$$

Next, the backward pass is applied, giving:

$$\begin{cases} r_{1,1} = r_{2,1}\mathbf{J}_2 + r_{2,2}\mathbf{H}_{2i} \\ r_{1,2} = r_{2,2}\mathbf{J}_2 \end{cases} \tag{1.29}$$

The solution of the recurrence equation is given by

$$r_k = \text{Dual}\left(\frac{\partial f}{\partial s_k} \,\middle|\, \frac{\partial^2 f}{\partial s_k \partial x_i}\right) \tag{1.30}$$

### 1.7.5 Reverse on reverse

**Mathematically**

$$\nabla^2 f(x) = J_{\nabla f}(x) \tag{1.31}$$

Equivalently, for a given direction $w$, it computes the Vector-Hessian Product (VHP)

$$D(\nabla f)^*[w] = H^T w = Hw \tag{1.32}$$

The last equation stands due to the symmetric nature of the Hessian matrix. Repeating this process for every basis $w = e_i$ yields the full Hessian.

**Order**

The reverse-on-forward strategy applies the reverse-mode to the reverse-mode computation of the gradient.

1. A forward pass to obtain function's values
2. A first reverse pass to compute first-order adjoints
3. A second reverse pass to compute second-order adjoints

**Forward pass (values computation)**

Let the primal variables

$$s_0 = x, \quad s_k = fk(s_{k-1}), \quad k = 1, \dots K \tag{1.33}$$

During the forward pass, the $K$ intermediary values are stored, as in standard reverse-mode AD.

**First reverse pass (gradient computation)**

Define the first-order adjoints

$$r_k \triangleq \frac{\partial f}{\partial s_k} \tag{1.34}$$

They satisfy the recurrence equations obtained above

$$\begin{cases} r_K = 1, \\ r_{k-1} = r_k \mathbf{J}_k \end{cases} \tag{1.35}$$

with $\mathbf{J}_k = \frac{\partial f_k}{\partial s_{k-1}}$.

At the end of the pass, we obtain $\nabla f(x) = r_0$. All adjoints are stored, as they are required for the second reverse pass.

**Second reverse pass (second-order adjoints)**

The adjoint equations are differentiates w.r.t themselves.

Let $\dot{r}_k$ denote the *second-order adjoint* defined by

$$\dot{r}_k \triangleq \frac{\partial}{\partial x_i}\left(\frac{\partial f}{\partial s_k}\right) \tag{1.36}$$

for a fixed input direction $e_i$ (one-hot encoding of $x_i$).

Differentiating the first-order recurrence

$$r_{k-1} = r_k \mathbf{J}_k \tag{1.37}$$

w.r.t $x_i$ yields

$$\dot{r}_{k-1} = \dot{r}_k \mathbf{J}_k + r_k \frac{\partial \mathbf{J}_k}{\partial x_i} \tag{1.38}$$

Using the chain rule,

$$\frac{\partial \mathbf{J}_k}{\partial x_i} = \frac{\partial \mathbf{J}_k}{\partial s_{k-1}} \frac{\partial s_{k-1}}{\partial x_i} = \mathbf{H}_{ki} \tag{1.39}$$

where $\mathbf{H}_{ki}$ denotes the Hessian of $f_k$ contracted with the forward sensitivity $\frac{\partial s_{k-1}}{\partial x_i}$.

The second-order reverse recurrence becomes

$$\dot{r}_{k-1} = \dot{r}_k \mathbf{J}_k + r_k \mathbf{H}_{ki} \tag{1.40}$$

with the initial condition $\dot{r}_K = 0$ since $r_K = 1$ is constant.

After completing the second reverse pass, the Hesian column corresponding to direction $e_i$ is obtained as

$$\nabla^2 f(x) e_i = \dot{r}_0 \tag{1.41}$$

Repeating the process for all $i = 1, \dots n$ yields the full Hessian matrix.

### 1.7.6 Comparison

| Strategy | Object computed | Output per run | Passes | Time cost | Memory cost |
|---|---|---|---|---|---|
| FoF | $w^T Hv$ | Scalar | $1 \rightarrow$ | $\mathcal{O}(K)$ | Low |
| FoR | $Hv$ | $\mathbb{R}^n$ | $1 \rightarrow +1 \leftarrow$ | $\mathcal{O}(K)$ | Moderate |
| RoF | $Hw$ | $\mathbb{R}^n$ | $1 \rightarrow +1 \leftarrow$ | $\mathcal{O}(K)$ | Moderate |
| RoR | $\nabla^2 f$ (column-wise) | $\mathbb{R}^n$ | $1 \rightarrow +2 \leftarrow$ | $\mathcal{O}(K)$ | Very high |

Table 1.1: Comparison of second-order AD strategies ($\rightarrow$: forward, $\leftarrow$: backward)

| Criterion | FoF | FoR | RoF | RoR |
|---|---|---|---|---|
| Computes full Hessian directly | No | No | No | No |
| Produces Hessian-vector product | No | Yes | Yes | Yes |
| Uses dual numbers | Yes | Yes | Yes | No |
| Requires primal trace storage | No | Yes | Yes | Yes |
| Requires adjoint storage | No | Yes | Yes | Yes |
| Requires second derivatives of primitives | Yes | Yes | Yes | Yes |
| Exploits Hessian symmetry | No | Yes | Yes | Yes |
| Suitable for large $n$ | No | Yes | Yes | No |
| Implementation complexity | Low | Medium | Medium | Very high |

Table 1.2: Feature comparison of Hessian computation strategies

| Strategy | Number of runs | Total cost |
|---|---|---|
| FoF | $n^2$ | $\mathcal{O}(n^2 K)$ |
| FoR | $n$ | $\mathcal{O}(nK)$ |
| RoF | $n$ | $\mathcal{O}(nK)$ |
| RoR | $n$ | $\mathcal{O}(nK)$ (large constant) |

Table 1.3: Cost of assembling the full Hessian matrix

Among the four strategies, forward-on-reverse is generally preferred in practice, as it provides Hessian-vector products at a computational cost comparable to that of a gradient evaluation, while avoiding the prohibitive memory requirements of reverse-on-reverse differentiation.

## 1.8 Implicit differentiation

Above sections considered *explicit* functions where each intermediary step was known. *Implicit* functions differentiation is considered when the intermediary steps are hidden and only the output is known (e.g. Newton's method, Gradient Descent).

Usual AD is inefficient for this type of functions due to the requirements of unrolling the implicit function behavior. Instead, one could use the fact that the derivatives depends on the solution and the path.

For instance, one may consider the square root function $x = \sqrt{a}$ and extract its implicit formulation through a fixed-point equation.

$$\begin{aligned} x &= \sqrt{a} \\ x^2 &= a \\ 2x^2 &= x^2 + a \\ g(x,a) = x &= \frac{1}{2}\left(x + \frac{a}{x}\right) \end{aligned} \tag{1.42}$$

Iterating, using $x_{k+1} = g(x_k, a)$, from an initial guess $x_0$, the solution would eventually converge to $x^* = \sqrt{a}$. The implicit equation is $F(x,a) = x - g(x,a)$ and its solution satisfies $F(x^*, a) = 0$.

**Theorem 1.1** (Inverse function theorem)**.** Assume

- $f : \mathcal{W} \to \mathcal{W}$ is $C^2$
- $\partial f(w_0)$ is invertible.

Then

- $f$ is bijective from a neighborhood of $w_0$ to a neighborhood of $f(w_0)$
- For $\omega$ in a neighborhood of $f(w_0)$, $f^{-1}$ is $C^2$ and $\partial f^{-1}(\omega) = (\partial f(f^{-1}(w)))^{-1}$ where the last equation has been obtained through the chain rule.

**Theorem 1.2** (Implicit function theorem (IFT, univariate case))**.** Assume

- $F : \mathbb{R} \to \mathbb{R}$
- $\exists (w_0, \lambda_0)$ such that $F(w_0, \lambda_0) = 0$ and $\partial_1 F(w_0, \lambda_0) \neq 0$
- $F(w, \lambda)$ is $C^2$ in a neighborhood $\mathcal{U}$ of $(w_0, \lambda_0)$

Then there exists a neighborhood $\mathcal{V} \subseteq \mathcal{U}$ where exists $w^*(\lambda)$ such that:

$$\begin{aligned} w^*(\lambda_0) &= w_0 \\ F(w^*(\lambda), \lambda) &= 0, \quad \forall (w^*(\lambda), \lambda) \in \mathcal{V} \\ \partial w^*(\lambda) &= -\left(\partial_1 F(w^*(\lambda), \lambda)\right)^{-1} \partial_2 F(w^*(\lambda), \lambda) \end{aligned} \tag{1.43}$$

**Example 1.1.** Implicit relation between $x$ and $y$

$$x^2 + y^2 = 1$$

Two possible explicit functions

$$\begin{aligned} y^+(x) &= \sqrt{1 - x^2} \\ y^-(x) &= -\sqrt{1 - x^2} \end{aligned}$$

Let $F(y, x) = x^2 + y^2 - 1$. Given an initial point $(x_0, y_0)$ with $x_0^2 + y_0^2 = 1$.

- if $y_0 > 0$ then IFT holds and $y^*(x) = y^+(x)$,
- if $y_0 < 0$ then IFT holds and $y^*(x) = y^-(x)$,

- if $y_0 = 0$ then $\partial_1 F(y_0, x_0) = 2y_0 = 0$ so IFT does not hold

This example shows that even a function such as $F(y, x) = x^2 + y^2 - 1$ with 2 possible explicit functions does not violate the IFT.

**Theorem 1.3** (Implicit function theorem (IFT, multivariate case))**.** Assume

- $F : \mathcal{W} \times \Lambda \to \mathcal{W}$
- $\exists (w_0, \lambda_0)$ such that $F(w_0, \lambda_0) = 0$ and $\partial_1 F(w_0, \lambda_0) \neq 0$
- $F(w, \lambda)$ is $C^2$ in a neighborhood $\mathcal{U}$ of $(w_0, \lambda_0)$

Then there exists a neighborhood $\mathcal{V} \subseteq \mathcal{U}$ where exists $w^*(\lambda)$ such that:

$$\begin{aligned} w^*(\lambda_0) &= w_0 \\ F(w^*(\lambda), \lambda) &= 0, \quad \forall (w^*(\lambda), \lambda) \in \mathcal{V} \\ \partial w^*(\lambda) &= -\left(\partial_1 F(w^*(\lambda), \lambda)\right)^{-1} \partial_2 F(w^*(\lambda), \lambda) \end{aligned} \tag{1.44}$$

### 1.8.1 Implicit JVP and VJP

For implicit differentiation, derivatives must be propagated differently.

Consider the implicit function defined by $F(w^*, \lambda) = 0$, from which one may want to compute the JVP and VJP.

#### JVP

Reminding that the forward tangent a step $k$, $t_k$, is computed using the known previous tangent $t_{k-1}$, and the function at the step, assuming the IFT holds, the following equality may be derived.

$$t_k = -\left(\partial_1 F(w^*, \lambda)\right)^{-1} \cdot \partial_2 F(w^*, \lambda) \cdot t_{k-1} \tag{1.45}$$

where $t_k = \partial w^*(\lambda)/\partial s_0$ and $t_{k-1} = \partial \lambda / \partial s_0$. Each JVP of the implicit function may be obtaine by solving a linear system.

Setting $A = -\partial_1 F(w^*(\lambda), \lambda)$ and $B = \partial_2 F(w^*(\lambda), \lambda)$, the system to solve becomes:

$$t_k = A^{-1} B t_{k-1} \tag{1.46}$$

Once the forward pass is done, $A$ is fixed and the linear system can be solved efficiently for each JVP by using the LU decomposition of $A$.

$$\begin{aligned} LU t_k &= B t_{k-1} \\ &\Downarrow \\ \begin{cases} Ly &= B t_{k-1} \\ U t_k &= y \end{cases} \end{aligned} \tag{1.47}$$

**VJP**

Computing the VJP boils down to the computation of the reverse tangent $r_{k-1}$ knowing the next reverse tangent $r_k$ and the function at this step. Assuming the IFT holds, the following equality can be derived.

$$r_{k-1} = -\left(\partial_2 F(w^*, \lambda)\right)^T \cdot \left(\partial_1 F(w^*, \lambda)\right)^{-T} \cdot r_k \tag{1.48}$$

where $r_{k-1} = \partial s_0 / \partial \lambda$ and $r_k = \partial s_0 / \partial w^*(\lambda)$. Denoting $A = \partial_1 F(w^*(\lambda), \lambda)$ and $B = \partial_2 F(w^*(\lambda), \lambda)$, the VJP can also be obtained as the solution of a linear system.

$$\begin{cases} A^T y = r_k \\ r_{k-1} = -B^T y \end{cases} \tag{1.49}$$

### 1.8.2 AD with optimization problem

Consider the optimization problem defined by:

$$\min c^T x \quad \text{s.t.} \quad Ax = b, \quad x \geq 0 \tag{1.50}$$

and its dual:

$$\max b^T y \quad \text{s.t.} \quad A^T y \leq c \tag{1.51}$$

The KKT conditions gives the optimality conditions for this problem:

$$\begin{cases} Ax = b \\ (A^T y - c) \perp x \geq 0 \end{cases} \tag{1.52}$$

The KKT conditions can be rewritten as an implicit function $F((x, y), (A, b, c))$:

$$F((x, y), (A, b, c)) = \begin{bmatrix} Ax - b \\ Diag(x)(A^T y - c) \end{bmatrix} \tag{1.53}$$

If the IFT holds for this function, then $\partial_1 F(w^*, \lambda)$ can be computed.

$$\frac{\partial F}{\partial (x, y)} = \begin{bmatrix} A & 0 \\ Diag(A^T y - c) & Diag(x) A^T \end{bmatrix} \tag{1.54}$$

## 1.9 Sparse AD

Often in practice, the Jacobian matrix is sparse, meaning that many of its entries are zero. In such cases, sparse AD techniques can be used to exploit the sparsity pattern and reduce the computational cost of the Hessian. For this section, the sparsity pattern of the Jacobian matrix is assumed to be known.

Consider for instance the following Jacobian matrix:

$$J = \begin{bmatrix} 0 & * & 0 & * & 0 \\ 0 & 0 & 0 & * & * \\ 0 & * & * & 0 & 0 \\ * & 0 & * & 0 & 0 \end{bmatrix} \tag{1.55}$$

Computing the Hessian using the standard AD techniques would require 5 JVP or 4 VJP. It is possible to reduce the cost by grouping rows or columns together as long as they do not share any non-zero entry. For instance, with the Jacobian matrix above, the Hessian can be computed with only 2 JVP or 2 VJP.

The intuition comes from the fact that when we compute a JVP or a VJP, the zero entries do not contribute to the result. Hence, columns or rows can be grouped together if they do not share non-zero entries.

For instance, computing the first two JVPs yields

$$J\begin{bmatrix}1\\0\\0\\0\\0\end{bmatrix} = \begin{bmatrix}0\\0\\0\\*\end{bmatrix} \qquad \text{and} \qquad J\begin{bmatrix}0\\1\\0\\0\\0\end{bmatrix} = \begin{bmatrix}*\\0\\*\\0\end{bmatrix} \tag{1.56}$$

Combining these two JVPs gives:

$$J\begin{bmatrix}1\\1\\0\\0\\0\end{bmatrix} = \begin{bmatrix}*\\0\\*\\*\end{bmatrix} \tag{1.57}$$

Using the known sparsity pattern, the results can be put back in the right place. This process is called *decompression*.

Hence computing the Hessian boilds down to combining the following columns and using them as vectors for the JVPs:

$$\begin{bmatrix}0 & * & 0 & * & 0\\0 & 0 & 0 & * & *\\0 & * & * & 0 & 0\\* & 0 & * & 0 & 0\end{bmatrix} \quad \Rightarrow \quad v_1 = \begin{bmatrix}1\\1\\0\\0\\1\end{bmatrix}, \quad v_2 = \begin{bmatrix}0\\0\\1\\1\\0\end{bmatrix} \tag{1.58}$$

Or for the VJP:

$$\begin{bmatrix}0 & * & 0 & * & 0\\0 & 0 & 0 & * & *\\0 & * & * & 0 & 0\\* & 0 & * & 0 & 0\end{bmatrix} \quad \Rightarrow \quad v_1 = \begin{bmatrix}1 & 0 & 0 & 1\end{bmatrix}, \quad v_2 = \begin{bmatrix}0 & 1 & 1 & 0\end{bmatrix} \tag{1.59}$$

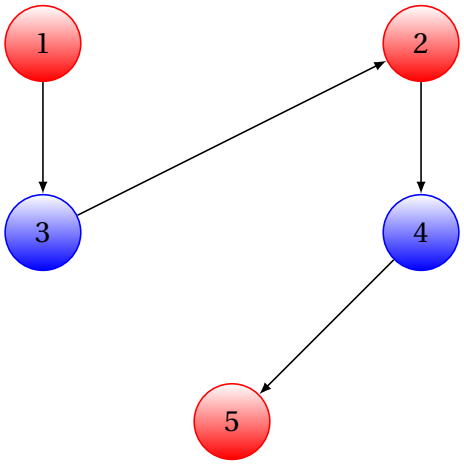
### 1.9.1 Sparsity detection

The problem of grouping the columns or rows of a sparse matrix can be modeled as a graph coloring problem in many ways. For instance, it could be a graph where each column is a node and where an edge connects two nodes if they share a non-zero entry.

Once the problem is modeled, any graph coloring algorithm can be used to color the graph such that no two adjacent nodes have the same color. As a result, each color represents a group of columns that can be combined together for the JVPs. The same can be done for the rows for the VJPs.

Here is a visual example.

$$J = \begin{bmatrix} 0 & * & 0 & * & 0 \\ 0 & 0 & 0 & * & * \\ 0 & * & * & 0 & 0 \\ * & 0 & * & 0 & 0 \end{bmatrix} \tag{1.60}$$



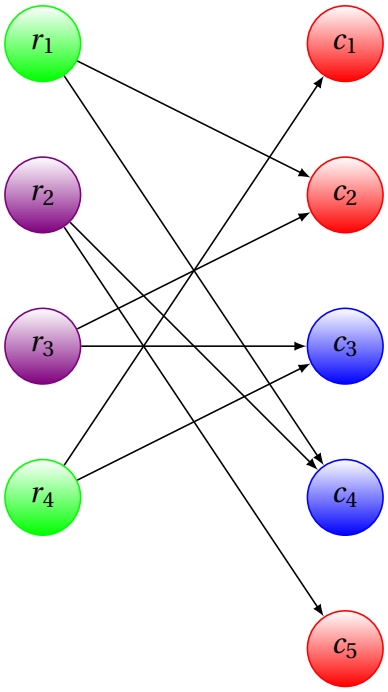

In more complex sparsity patterns, somes nodes could be linked multiple times, and therefore a particular care must be taken to ensure no multipy linked nodes share an edge.

$$J = \begin{bmatrix} 0 & 0 & * \\ * & * & 0 \\ * & * & 0 \end{bmatrix} \tag{1.61}$$

Node 1 and 2 should be linked by two edges because they share two non-zero entries, hence the colors must be different.

The probem can also be modeled as a bipartite graph, where one set of nodes represents the columns and the other set represents the rows. There will be an edge between a column node and a row node if the corresponding entry in the matrix is non-zero. Considering the same instance

$$J = \begin{bmatrix} 0 & * & 0 & * & 0 \\ 0 & 0 & 0 & * & * \\ 0 & * & * & 0 & 0 \\ * & 0 & * & 0 & 0 \end{bmatrix} \quad or \quad \begin{bmatrix} 0 & * & 0 & * & 0 \\ 0 & 0 & 0 & * & * \\ 0 & * & * & 0 & 0 \\ * & 0 & * & 0 & 0 \end{bmatrix} \tag{1.62}$$




With this modeling, the coloring problem is a distance-2 graph coloring problem. Two column nodes being at distance 2 from each other can't share the same color because if they share a non-zero entry ar row $n$, they will be connected to the same row node $n$.

The same applies for the rows.

### 1.9.2 Symmetric matrix

When the Jacobian matrix is symmetric, a more efficient algorithm can be used to compute the HVP. For instance, consider the following symmetric Jacobian matrix.

$$J = \begin{bmatrix} a_1 & a_2 & a_3 & 0 \\ a_2 & a_4 & 0 & a_5 \\ a_3 & 0 & a_6 & 0 \\ 0 & a_5 & 0 & a_7 \end{bmatrix} \tag{1.63}$$

With the previous sparse AD techniques, 3 HVPs or 3 VHPs would be required, instead of 4 wit the standard AD. Using the symmetry of the matrix, this computations can be reduced to 2 HVPs or 2 VHPs. Indeed, columns can colored as

$$J = \begin{bmatrix} a_1 & a_2 & a_3 & 0 \\ a_2 & a_4 & 0 & a_5 \\ a_3 & 0 & a_6 & 0 \\ 0 & a_5 & 0 & a_7 \end{bmatrix} \tag{1.64}$$

Using the symmetry property, the entry shared between columns 2 and 3 can be ignored since this value would be also computed by the HVP on the first column. Hence, it can be considered as 0 for now, and replaced back later.

Computing the two HVPs and retrieving the full Hessian matrix shows

$$J \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 + a_3 \\ a_2 + a_5 & a_4 \\ a_3 & a_6 \\ a_7 & a_5 \end{bmatrix} \tag{1.65}$$

$a_2$ can be obtained by solving a small linear system.

Using computed values and knowing the sparsity pattern, the full matrix can be reconstructed.

The same process can be developed for VHPs by grouping rows instead of columns.

### 1.9.3 Star coloring

Given a graph $G = (V, E)$, and a coloring function $\phi : V \rightarrow \{1, \ldots, p\}$, a star coloring is distance-1 coloring with the additional constraint that every path of 4 vertices uses at least 3 colors. In other words, no path of length 3 is *bi-chromatic*. This prevents *length-3* dependency chains. For instance, considering the same symmetric Jacobian matrix as above

$$J = \begin{bmatrix} a_1 & a_2 & a_3 & 0 \\ a_2 & a_4 & 0 & a_5 \\ a_3 & 0 & a_6 & 0 \\ 0 & a_5 & 0 & a_7 \end{bmatrix} \tag{1.66}$$

Coloring its associated graph with the classic distance-1 coloring would yield the same matrix as (1.64). Computing the HVPs, one would obtain the matrix (1.65) and solving a linear system would be required to retrieve $a_2$.

Usings star coloring, one could retrieve explicitly all the element.

Consider the following coloring

$$J = \begin{bmatrix} a_1 & a_2 & a_3 & 0 \\ a_2 & a_4 & 0 & a_5 \\ a_3 & 0 & a_6 & 0 \\ 0 & a_5 & 0 & a_7 \end{bmatrix} \tag{1.67}$$

Using the green color ensures the star coloring condition. Now, computing the 3 HVPs

$$J \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 + a_3 & 0 \\ a_2 & a_4 & a_5 \\ a_3 & a_6 & 0 \\ 0 & a_5 & a_7 \end{bmatrix} \tag{1.68}$$

All the values can be immediately retrieved from the matrix without solving any linear equation system.

This proves its better efficiency compared to the usual distance-1 coloring (4 HVPs), and the symmetric distance-1 coloring (2 HVPs + linear system).

### 1.9.4 Acyclic coloring

Given a graph $G = (V, E)$, and a coloring function $\phi : V \rightarrow \{1, \ldots, p\}$, an acyclic coloring is a distance-1 coloring with the additional constraint that every cycle uses at least 3 colors. This allows to form forests of disjoints trees with the subgraph of each color pair.

TODO better explain HVP computation with acyclic coloring and give an example

### 1.9.5 Chromatic number ($\xi$)

For every graph G,

$$\xi_1(G) \leq \xi_{acyclic}(G) \leq \xi_{star}(G) \leq \xi_2(G) \tag{1.69}$$

$\xi_1$ is the usual distance-1 chromatic number, $\xi_2$ the distance-2 chromatic number.

# Neural networks

Neural networks are a class of machine learning models inspired by the structure and function of the human brain. They are composed of layers of interconnected nodes (neurons) that process and transmit information.
First let's define some variables:

- $X$: input data (matrix)
- $y$: target data
- $W_k$: weights matrix at layer $k$
- $b_k$: bias vector at layer $k$
- $\sigma$: activation function (ReLU, sigmoid, etc)
- $\ell(.)$: loss function
- $H$: number of hidden layers
- $S_i$: intermediate state

To propagate and update the information through the network we use forward pass and backward pass and so automatic differentiation.
We can describe the forward pass of a neural network in two equivalent ways:

$$\begin{array}{ll} \text{Right to left:} & \text{Left to right:} \\ S_0 = X & S_0 = x \\ S_{2k-1} = W_k S_{2k-2} + b_k & S_{2k-1} = S_{2k-2} W_k + b_k \\ S_{2k} = \sigma(S_{2k-1}) & S_{2k} = \sigma(S_{2k-1}) \\ S_{2H+1} = W_{k+1} S_{2H} & S_{2H+1} = S_{2H} W_{k+1} \\ S_{2H+2} = \ell(S_{2H+1}, Y) & S_{2H+2} = \ell(S_{2H+1}, Y) \end{array} \tag{2.1}$$

The principal differences is that the weights acts on the left or on the right of the data, it can be useful depending whether you represents inputs as rowvectors or columnvectors.
It can be represented by this computational graph:

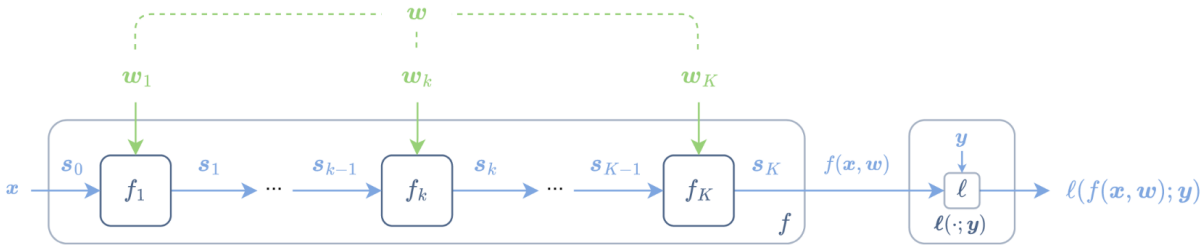



Figure 2.1: Neural network forward pass

And the backward pass can be represented like this:

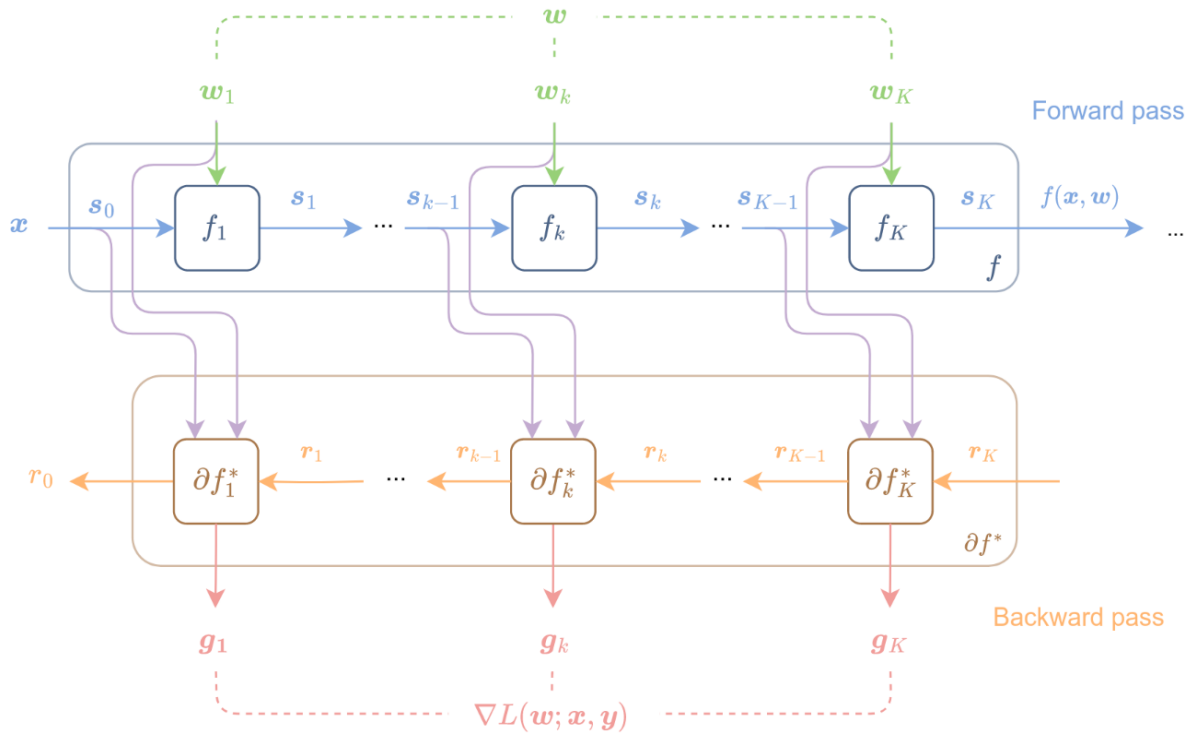



Figure 2.2: Neural network backward pass

## 2.1 Autoregressive models

An Autoregressive model wants to predict the next values in a sequence based on its previous values. Given a sequence of $n_{cxt}$ (context) past vectors $x_{-1}, x_{-2}, \ldots, x_{-n_{cxt}}$, the model aims to predict the next vector $x_0$ and maybe more $x_1$. Example, we want to predict $x_0$ and $x_1$ based on the past:

$$\begin{aligned} p(x_0, x_1 | x_{-1}, \ldots, x_{-n_{cxt}}) &= p(x_0 | x_{-1}, \ldots, x_{-n_{cxt}}) \cdot p(x_1 | x_0, x_{-1}, \ldots, x_{-n_{cxt}+1}, x_{-n_{cxt}}) \\ &\approx p(x_0 | x_{-1}, \ldots, x_{-n_{cxt}}) \cdot p(x_1 | x_0, x_{-1}, \ldots, x_{-n_{cxt}+1}) \end{aligned} \tag{2.2}$$

So the models aims to predict the probability distribution of the next vector $x_0$ with $\hat{p}(x_0|X)$ with $X$ that concatenates all the context vectors. And then we use the cross-entropy to measure how well the predicted distribution $\hat{p}$ matches the true distribution $p$:

$$\mathscr{L}(X) = -\sum_{x_0} p(x_0|X) \log(\hat{p}(x_0|X)) \tag{2.3}$$

## 2.2 Tokenization

How can we use this autoregressive model for LLM (Large Language Models) for example ? Consider that we have $n_{cxt}$ characters of context and we want to predict the next characters, this means that we would have a one-hot encoding in $\mathbb{R}^{26}$, which means $n_{voc} = 26$. This means that $n_{cxt}$ should be a very large number, but this is annoying because transormers have a quadratic complexity in $n_{cxt}$.
Another idea could be to turn each word into a one-hot encoding, but the vocabulary size is often very large (+200k words), with this we could have a relative low $n_{ctx}$, but $n_{voc}$ would be to big.
An intermediate solution is to use byte pair encoding algorithm which greedily merges the most frequent pairs of characters into new tokens.
Example: consider the word "abracadabra", we see that we have two frequent pairs "ab" and "ra", so we can merge them into new tokens $X$ and $Y$ respectively:

$$\text{abracadabra} \to \text{XYcadXY} \tag{2.4}$$

then we can repeat the process until we reach the desired vocabulary size.
The next iteration could give:

$$\text{XYcadXY} \to \text{ZcadZ} \tag{2.5}$$

This tokenization method allows us to have good trade-off between the vocabulary size $n_{voc}$ and the context size $n_{cxt}$. But if the model isn't trained enough we could have an issue which is that the model doesn't output fully constructed words, for example it could output half of a word because that what makes more sense even though the word is half complete.

## 2.3 Embedding

Consider a vocabulary of size $n_{voc}$, a bigram model and a network with $d$ layers. The model would be:

$$\hat{p}(x_0|x_{-1}) = \text{softmax}(W_d \tanh(\dots \tanh(W_1 x_{-1}) \dots)) \tag{2.6}$$

The matrix $W_1$ has $n_{voc}$ columns and $W_d$ has $n_{voc}$ rows, so when $n_{voc}$ is large, the model becomes very big.
So an idea would be to use an encoder and a decoder to reduce the sizes, it is called an embedding size $d_{emb} \ll n_{voc}$. The encoder ($C \in \mathbb{R}^{d_{emb} \times n_{voc}}$) maps the one-hot encoding of size $n_{voc}$ to a dense vector of size $d_{emb}$ and the decoder ($D \in \mathbb{R}^{n_{voc} \times d_{emb}}$) maps back the dense vector to a vector of size $n_{voc}$. So the model becomes:

$$\hat{p}(x_0|x_{-1}) = \text{softmax}(DW_d \tanh(\dots \tanh(W_1 C x_{-1}) \dots)) \tag{2.7}$$

If we choose wisely $d_{emb}$, which means much smaller than $n_{voc}$, then it is faster to compute $W_1(Cx_{-1})$ than in the previous model. Moreover we are forcing $W_1 C$ to be low-rank which can help to reduce overfitting but reduce the expressivness (capacity to capture a range of possible relation between input and output) of the model. It is useful to share the embedding between different input and output when $n_{ctx} > 1$, we also found that forcing $D = C^T$ appears to work well in practice.

### 2.3.1 Shared embedding

Let's investigate the case where we have $n_{cxt} > 1$. When $n_{cxt} > 1$, the encoder C is shared by all tokens, so we get this model:

$$\hat{p}(x_0|x_{-1},\dots,x_{-n_{cxt}}) = \text{softmax}(DW_d \tanh(\dots \tanh(W_1 \begin{bmatrix} Cx_{-1} \\ \vdots \\ Cx_{-n_{cxt}} \end{bmatrix})\dots)) \tag{2.8}$$

We can note that now $W_1$ has $n_{cxt}d_{emb}$ columns. Assuming $d_{emb} \ll n_{voc}$ and $n_{cxt} \gg 1$, this is much smaller than the $n_{ctx}n_{voc}$ that we had before the embedding. The number of rows of $W_2$ is not affected by the embedding so it remains $n_{voc}$.
We could represent this model like this:

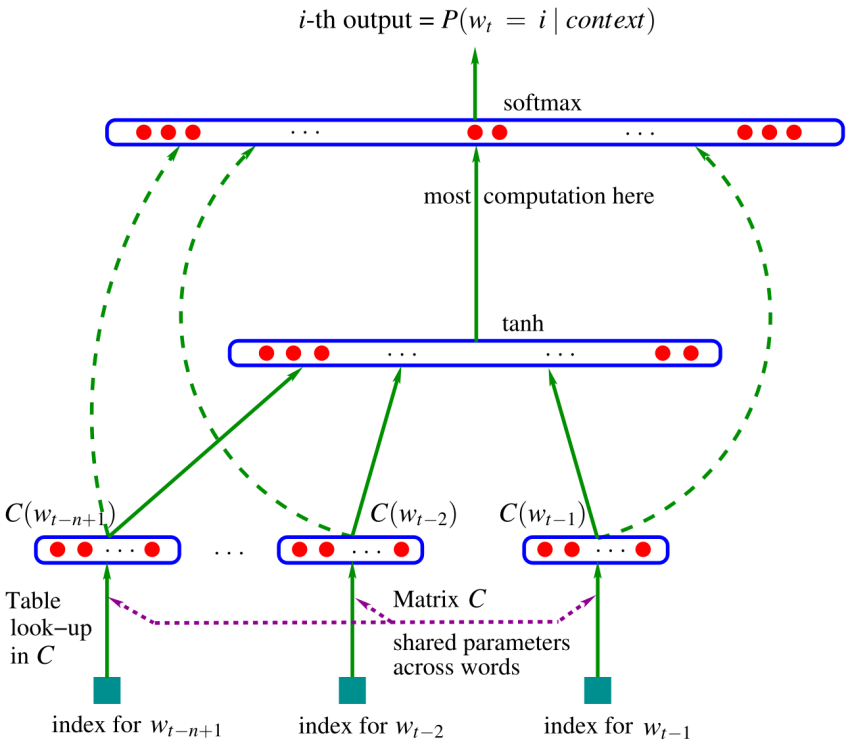



Figure 2.3: Neural network with shared embedding

## 2.4 Recurrent neural networks (RNN)

A RNN is a type of neural network that is designed to process sequential data by taking advantages of the memory of previous inputs. The idea is to have a hidden state $h_t$ that depends on the current input $x_t$ and the previous hidden state $h_{t-1}$. With this idea, the network reuses the same weights at every time step and thus gives meaning to the sequence. The equations of a simple RNN are:

$$\begin{cases} h_{t+1} = \tanh(Wh_t + Ux_{t-1} + b) \\ \hat{y}_t = \text{softmax}(Vh_t + c) \end{cases} \tag{2.9}$$

Where $W, U, V$ are weight matrices and $b, c$ are bias vectors.
This model presents some limitations, for example it is difficult to learn long-term dependencies because of the vanishing/exploding gradient problem. The fact that it processes a

sequence step by step makes it difficult to parallelize the computations. The fact that it need to store the hidden state at each time step can be also memory-intensive for long sequences.

## 2.5 Attention head

First we need to define a numerical dictionary. Consider keys $k_i \in \mathbb{R}^{d_k}$, values $v_i \in \mathbb{R}^{d_v}$. Given a query $q \in \mathbb{R}^{d_k}$, we want to retrieve the value $v_i$ corresponding to the key $k_i$ that is the most similar to the query $q$. To do this, we can use the dot-product. With that definition, we can define the attention head:

$$\text{Attention}(Q,K,V) = \sum_{i=1}^{n_{ctx}} \alpha_i v_i \tag{2.10}$$

where $\alpha = \text{softmax}(< q,k_1 >, \cdots, < q,k_{n_{ctx}} >)$ is the attention weight for key $k_i$. But in practice, we have vectors of queries, keys and values, each contained in matrices $Q,K,V$. So we can rewrite the attention head as:

$$\text{Attention}(Q,K,V) = V\text{softmax}\left(\frac{K^T Q}{\sqrt{d_k}}\right) \tag{2.11}$$

Where the division by $\sqrt{d_k}$ is used to prevent the dot-product from growing too large.
For a transformers, we could explain $Q,K,V$ like this, for each token we have a vector $q,k,v$ that represent:

- $q$: What am I looking for?
- $k$: What do I contain?
- $v$: What information do I pass if IâĂŹm selected?

### 2.5.1 Masked attention

To ensure that the model only attends to previous tokens and not future tokens, we can use a masked attention mechanism. This is done by applying a mask to the attention weights before the softmax operation. The mask is typically a lower triangular matrix that sets the weights corresponding to future tokens to negative infinity, effectively preventing them from contributing to the attention output.
So with masked attention, the equation becomes:

$$\begin{cases} M = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ -\infty & 0 & \ddots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ -\infty & -\infty & \cdots & 0 \end{bmatrix} \\ \text{Attention}(Q,K,V) = V\text{softmax}\left(M + \frac{K^T Q}{\sqrt{d_k}}\right) \end{cases} \tag{2.12}$$

### 2.5.2 Multi-head attention

Instead of having a single attention head, we can have multiple attention heads that can understand different aspects of the input data. Each head has its own set of weights and computes its own attention output, all in parallel. After computing all the heads, their outputs are concatenated and linearly transformed to be in the right dimension with $W^O$, this is thus gathering all of the meanings of the attention different heads to a single representation. So the multi-head attention can be defined as:

$$\begin{cases} \text{head}_j = \text{Attention}(W_j^Q Q, W_j^K K, W_j^V V) \\ \\ \text{MultiHead}(Q,K,V) = W^O \begin{bmatrix} \text{head}_1 \\ \text{head}_2 \\ \vdots \\ \text{head}_h \end{bmatrix} \end{cases} \tag{2.13}$$

## 2.6 Decoder-only transformer

A decoder-only transformer is a type of transformer architecture that is used for autoregressive model. It consists of reading the tokens and predicting the next token in the sequence. In this type of transformer, the matrices $Q, K, V$ are the same, and we define them as the input tokens $CX$, $C$ being the embedding matrix. So the decoder-only transformer can be computed with $\text{MultiHead}(CX, CX, CX)$. Let's seek the different mechanisms used in a decoder-only transformer.

### 2.6.1 Positional encoding

The positional encoding is essential in a transformer model because it provides information about the order of the tokens in the input sequence. Since transformers do not have a built-in notion of sequence order like RNNs, positional encodings are added to the input embeddings to give the model a sense of position.
We cannot use one-hot encoding for the position because the dimension would be $n_{ctx}$ and not $d_{emb}$ as expected. The classic approach is to use sines and cosines with an angle that depends on the position of the token and the embedding dimension. The positional encoding for a token at position $pos$ and with the embedding dimension index $i$ and embedding dimension $d_{emb}$ could be defined as:

$$\begin{cases} angle = \dfrac{pos}{10000^{2i/d_{emb}}} \\ PE[pos, 2i] = \sin\left(angle\right) \\ PE[pos, 2i+1] = \cos\left(angle\right) \end{cases} \tag{2.14}$$

And then we add this positional encoding to the input embeddings before doing the attention mechanism:

$$\text{MultiHead}(CX + PE, CX + PE, CX + PE) \tag{2.15}$$

### 2.6.2 Residual connection

The principle of residual connection is to add the input of a layer to its output before applying the layer normalization (next subsection). This helps to mitigate the vanishing gradient problem and help the network to learn because its learning is more controlled, because we remind it of the original input.

### 2.6.3 Layer normalization

The norm of the gradient increases exponentially with the number of layers, so to prevent this we can use layer normalization. This helps stabilize the training process and improve convergence, by rescaling the data before the activation function, which means that we control the mean and the variance of the data. There is two way of doing this normalization, the batch normalization and the layer normalization. The batch normalization is difficult and not very efficient to use in transformers, so we use layer normalization. It is done like this:

$$\text{LayerNorm}(y_i) = \gamma \frac{y_i - \mu_i}{\sigma_i} + \beta \tag{2.16}$$

With $\gamma$ and $\beta$ two learnable parameters, $\mu_i$ the mean of the elements of $y_i$ and $\sigma_i$ the standard deviation of the elements of $y_i$. Then we can add this in the transformer architecture like this:

$$\text{LayerNorm}(\text{MultiHead}(CX + PE, CX + PE, CX + PE) + (CX + PE)) \tag{2.17}$$

### 2.6.4 Feed-forward network (FFN)

The feed-forward network consists in doing this operation:

$$\text{FFN}(x) = W_2 \text{ReLU}(W_1 x + b_1) + b_2 \tag{2.18}$$

With $W_1 \in \mathbb{R}^{d_{ff} \times d_{emb}}$ and $W_2 \in \mathbb{R}^{d_{emb} \times d_{ff}}$, $d_{ff}$ being the dimension of the feed-forward network, generally $d_{ff} = 4d_{emb}$. This feed-forward network is applied to each token independently and identically. The feed-forward network is used to introduce non-linearity and increase the model's capacity to learn complex patterns in the data. It is used the memorize more complex information in the weights, which helps the model to better understand the complex relationships between tokens in the sequence.

### 2.6.5 Transformers variation

To investigate ?

## 2.7 Performances of transformers

Before analysing the complexity of a transformer, let us reminds the main dimension of the variables:

- $n_{ctx}$: context size (number of tokens in the input sequence)
- $n_{voc}$: vocabulary size (number of unique tokens)

- $d_{emb}$: embedding size (dimension of embeddings)
- $d_k, d_v$: key and value dimension (per head)
- $d_{ff}$: feed-forward network dimension
- $N$: number of layers

We can thus analyse the time complexity of a transformer step by step (ignoring the mebedding step):

1. Linear projections for $Q, K, V$ (with $W_j^Q, W_j^K, W_j^V$): $\mathcal{O}(d_k d_{emb} n_{ctx})$
2. Attention score computation ($K^T Q$): $\mathcal{O}\left((d_k + d_v) n_{ctx}^2\right)$
3. Output projection ($W^O$): $\mathcal{O}(d_v d_{emb} n_{ctx})$
4. FFN: $\mathcal{O}\left(d_{emb} d_{ff} n_{ctx}\right)$

This brings the total cost for a single layer to:

$$\mathcal{O}\left((d_k + d_v) n_{ctx}^2 + (d_k + d_v + d_{ff}) d_{emb} n_{ctx}\right) \tag{2.19}$$

To get the total cost for $N$ layers, we just need to multiply by $N$.

$$\mathcal{O}\left(N(d_k + d_v) n_{ctx}^2 + N(d_k + d_v + d_{ff}) d_{emb} n_{ctx}\right) \tag{2.20}$$

Knowing that generally $d_k \approx d_v \approx d_{emb} \approx d_{ff}$, we can simplify the complexity to:

$$\mathcal{O}\left(N d_{emb} n_{ctx}^2 + N d_{emb}^2 n_{ctx}\right) = \mathcal{O}\left(N d_{emb} n_{ctx} (n_{ctx} + d_{emb})\right) \tag{2.21}$$

Here we can see the two dominant terms of the complexity:

- $N d_{emb} n_{ctx}^2$: comes from the attention score computation, it is the most critical term when $n_{ctx}$ is large.
- $N d_{emb}^2 n_{ctx}$: comes from the FFN and the projection, it is the most critical term when $d_{emb}$ and thus for wide models.

We can also observe that the dimension of the vocabulary $n_{voc}$ does not appear in the complexity. It is because it is only involved in the embedding complexity.

# Diffusion Models

The objective of diffusion models is to generate data by learning how to reverse a gradual noising process, turning random noise into structured samples through iterative denoising.

## 3.1 Tweedie's formula

First, we need to understand why diffusion models can be trained by denoising and why predicting the noise is equivalent to learning a score. For that, we need to introduce Tweedie's formula. Consider that we observe a noisy version of a random variable $X$. The noisy observation is given by $Y = X + \sigma\varepsilon$, where $\varepsilon \sim \mathcal{N}(0,1)$. We want, given this noisy observation $Y = y$, to estimate the original variable $X$. The MMSE estimator is given by $\mathbb{E}[X|Y = y]$. Tweedie's formula states that:

$$\mathbb{E}[X|Y = y] = y + \sigma^2 \nabla_y \log f_Y(y) \tag{3.1}$$

where $f_Y(y)$ is the probability density function of the noisy observation $Y$. And thus $\nabla_y \log f_Y(y)$ is a score function that tells us in which direction the probability mass increases and thus where cleaner data is. We can also estimate the noise $\varepsilon$:

$$\mathbb{E}[\varepsilon|Y = y] = -\sigma \nabla_y \log f_Y(y) \tag{3.2}$$

And this means that if we can predict the noise, we can also estimate the score function and thus the denoised data. This is the principle used in diffusion models.

## 3.2 Langevin dynamics (sampling)

Langevin dynamics is a way to sample from a probability distribution ($p(y)$) when you only know its log-density gradient (the score), not the density itself. The idea is to iteratively update a sample by taking small steps in the direction of the score, while also adding some random noise to ensure exploration of the space. The update rule for Langevin dynamics is given by:

$$y_{k+1} = y_k + \delta_k \nabla_y \log p(y_k) + \sqrt{2\delta_k} w_k \tag{3.3}$$

with $w_k \sim \mathcal{N}(0,1)$ and $\delta_k$ is a small step size. By repeating this process many times, the samples $y_k$ will converge to the target distribution $p(y)$. Adding this noise is important because it helps the samples to explore the space more effectively and avoid getting stuck in local modes of the distribution. The Langevin dynamics is seen in diffusion models at each denoising step.

## 3.3 Score matching

This is the part "learning the score" of diffusion models. Here the idea is to fix a noise level $\sigma$ and to corrupt the data $X$ with Gaussian noise to get $Y = X + \sigma\varepsilon$, with $\varepsilon \sim \mathcal{N}(0,1)$. Then we want to minimize the error to learn the noisy data distribution ($p_{X+\sigma\varepsilon}$):

$$\mathbb{E}\left[\|\varepsilon_\theta(X+\sigma\varepsilon) - \varepsilon\|^2\right] \tag{3.4}$$

But with this, $\sigma$ needs to be scaled properly, otherwise the model will not learn well. If $\sigma$ is too small, then the support of $X + \sigma\varepsilon$ may not cover the whole space, and thus $\varepsilon_\theta(y)$ may be inaccurate. If $\sigma$ is too large, then the noise dominates the signal, and the model may struggle to learn meaningful patterns.

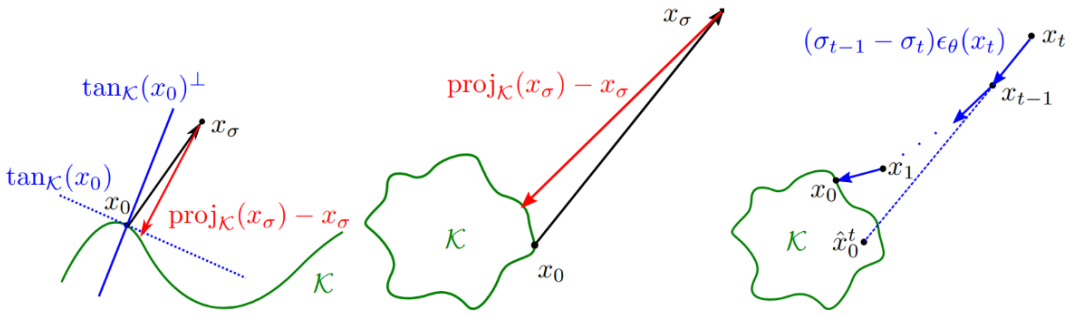
### 3.3.1 Variance dependent score

To address the issue of choosing the right noise level $\sigma$, we can introduce a variance-dependent score function. Instead of using a fixed $\sigma$, we can define a score function that depends on the noise level. The idea is to train a model $\varepsilon_\theta(y,\sigma)$ that takes both the noisy observation $y$ and the noise level $\sigma$ as inputs. The training objective becomes:

$$\mathbb{E}\left[\|\varepsilon_\theta(X+\sigma\varepsilon,\sigma) - \varepsilon\|^2\right] \tag{3.5}$$

## 3.4 Deterministic Sampling

On the picture below, we can see how does denoising looks like with DDIM (Denoising Diffusion Implicit Models). The idea is to use a deterministic process to reverse the diffusion process. If we denoise in one step, for low noise levels, it works quite well. For large noise levels, the score provides only a global direction, so multiple denoising steps are required to gradually approach the data manifold.




(a) Low noise (b) High noise (c) Denoising process (DDIM)

Figure 3.1: DDIM sampling example

DDIM move from a variance $\sigma_t$ to a variance $\sigma_{t-1}$, like in the figure above (c):

$$\begin{aligned}
X_t &= X_0 + \sigma_t\varepsilon \\
\mathbb{E}[X_{t-1}|X_t = x_t] &= \mathbb{E}[X_0|X_t = x_t] + \sigma_{t-1}\mathbb{E}[\varepsilon|X_t = x_t] \\
&= \mathbb{E}[X_t|X_t = x_t] - \sigma_t\mathbb{E}[\varepsilon|X_t = x_t] + \sigma_{t-1}\mathbb{E}[\varepsilon|X_t = x_t] \\
&= x_t - (\sigma_t - \sigma_{t-1})\mathbb{E}[\varepsilon|X_t = x_t] \\
&= x_t - (\sigma_t - \sigma_{t-1})\epsilon_\theta(x_t)
\end{aligned} \tag{3.6}$$

## 3.5 Denoising with randomness

Now we want to add some randomness to the denoising process, so we introduce DDPM (Denoising Diffusion Probabilistic Models). The idea is to add some noise during the denoising process to ensure diversity in the generated samples. Give $0 \leq \mu < 1$, pick $\sigma_{t'}$ such that $\sigma_{t-1} = \sigma_t^{\mu} \sigma_{t'}^{1-\mu}$ and we have the following sampler:

$$x_{t-1} = x_t + (\sigma_{t'} - \sigma_t)\epsilon_\theta(x_t, \sigma_t) + \eta w_t \tag{3.7}$$

with $w_t \sim \mathcal{N}(0,1)$. If we choose $\mu = 0$ and $\sigma_{t'} = \sigma_{t-1}$, we recover the DDIM sampler. And if we choose $\mu = 1/2$, we have the DDPM sampler. We have by assumption $\sigma_{t-1} < \sigma_t$. Then, because $\sigma_{t-1}$ is the geometric mean of $\sigma_t$ and $\sigma_{t'}$, it must be between so $\sigma_{t'} < \sigma_{t-1} < \sigma_t$. To choose $\eta$, we know we want $\sigma_{t-1}^2 = \mathbb{V}(\sigma_{t'}\varepsilon_\theta(x_t, \sigma_t)) + \eta W_t$. So we can choose $\eta = \sqrt{\sigma_{t-1}^2 - \sigma_{t'}^2}$.

### 3.5.1 Acceleration

We can accelerate the convergence of the previous sampler using the observation that "The noise prediction at a lower noise level is more accurate". We can thus define:

$$\bar{\epsilon}_t = \gamma\epsilon_\theta(x_t, \sigma_t) + (1-\gamma)\epsilon_\theta(x_{t+1}, \sigma_{t+1}) \qquad \gamma > 1 \tag{3.8}$$

## 3.6 Auto-Encoder

We want to find an encoder $E$ and a decoder $D$ that minimizes the loss:

$$\mathbb{E}[\|X - D(E(X))\|_2^2] \tag{3.9}$$

$E(X)$ typically reduces the dimension of $X$ to force the model to keep only essential features. If $E$ and $D$ are linear, then the auto-encoder is $x \to DEx$ and so only the product $DE$ matters and the problem becomes:

$$\min_{D,E} \mathbb{E}[\|X - DEX\|_F^2] \tag{3.10}$$

Because we want to compress the data, we have the constraint $\text{rank}(DE) \leq r$. So the auto-encoder is searching for the best rank-$r$ approximation of X. We can use the SVD of $X = U\Sigma V^T$ to find the best rank-$r$ approximation, and in our case it is working the same way as the PCA. And thus to approximate $X$ we can choose $DE = U_r \Sigma_r U_r^T$. With all this, an Auto-Encoder can be thought of as a nonlinear generalization of PCA.

### 3.6.1 Variationnal Auto-Encoder

We want to learn the distribution of our data represented by the random variable $X$. To improve learning, we can add artificial noise to learn a more interesting distribution.

$$\mathbb{E}[\|X - D(E_\mu(X) + \varepsilon \odot E_\sigma(X))\|_2^2] \tag{3.11}$$

### 3.6.2 Evidence Lower Bound (ELBO)

The evidence lower bound is a useful lower bound on the log-likelihood of some observed data. It is useful because it provides a guarantee on the worst-case for the log-likelihood of some distribution (e.g. $p(X)$) which models a set of data. But first we need to introduce the Kullback-Leibler (KL) divergence. The KL divergence is a measure of how one probability distribution diverges from a second, expected probability distribution. For two distributions $P$ and $Q$, the KL divergence from $Q$ to $P$ is defined as:

$$D_{KL}(P\|Q) = \sum_x P(x) \log\left(\frac{P(x)}{Q(x)}\right) \tag{3.12}$$

And with that we can define the ELBO.

$$\mathscr{L}(x) = \mathbb{E}[\log(f_{X|Z}(x|Y))] - D_{KL}((Y|X=x)\|Z) \tag{3.13}$$

The KL divergence is always non-negative, and measure the error made by our estimator.

### 3.6.3 Gaussian ELBO

We can compute the ELBO in the case of Gaussian distributions, like with the variationnal Auto-Encoder. Suppose $X = D_\mu(Z) + \varepsilon_1 \odot D_\sigma(Z)$ and $Y = E_\mu(X) + \varepsilon_2 \odot E_\sigma(X)$, with $\varepsilon_1, \varepsilon_2 \sim \mathcal{N}(0,1)$. Then we have:

$$2D_{KL}((Y|X=x)\|Z) = \|E_\mu(X)\|_2^2 + \|E_\sigma(X)\|_2^2 - \left(\sum_{i=1}^{r} \log((E_\sigma(X))_i^2) + r\right) \tag{3.14}$$

And for the first term of the ELBO:

$$\begin{aligned}
&\mathbb{E}\left[\log(f_{X|Z}(x|Y))\right] \\
&= \mathbb{E}\left[\log(f_{X|Z}(x|E_\mu(X) + \varepsilon_2 \odot E_\sigma(X)))\right] \\
&= -\frac{\log(2\pi)}{2} + \mathbb{E}\left[\|\mathrm{Diag}(D_\sigma(E_\mu(X) + \varepsilon_2 \odot E_\sigma(X)))^{-1}\left(x - D_\mu(D_\sigma(E_\mu(X) + \varepsilon_2 \odot E_\sigma(X)))\right)\|_2^2\right]
\end{aligned} \tag{3.15}$$

### 3.6.4 Monte-Carlo sampling

The gaussian ELBO can be estimated using Monte-Carlo sampling. Given $L$ samples $(\epsilon_1, \ldots, \epsilon_L)$ from $\mathcal{N}(0,1)$, it gives:

$$\mathbb{E}\left[\log(f_{X|Z}(x|Y))\right] \approx \frac{1}{L}\sum_{i=1}^{L} \log(f_{X|Z}(x|E_\mu(X) + \epsilon_i \odot E_\sigma(X))) \tag{3.16}$$

With the simple case where $D_\sigma(z) = 1$, we get this approximation ($L_2$ form):

$$\mathbb{E}\left[\log(f_{X|Z}(x|Y))\right] \approx -\frac{\log(2\pi)}{2} + \frac{1}{L}\sum_{i=1}^{L} \|x - D_\mu(E_\mu(x) + \epsilon_i)\|_2^2 \tag{3.17}$$

### 3.6.5 Maximum Likelihood Estimator with variationnal Auto-Encoder

Reminder, the encoder $E$ maps a data point $x$ to a Gaussian distribution $Y \sim \mathcal{N}(E_\mu(x), E_\sigma(x))$. The decoder $D$ maps a latent variable $z$ to the Gaussian distribution $\mathcal{N}(D_\mu)$. The maximum likelihood estimator, maximizes the following sum over our datapoints $x$ with its ELBO:

$$\sum_x \log(f_X(x)) \geq \sum_x -D_{KL}((Y|X=x)\|Z) + \mathbb{E}[\log(f_{X|Z}(x|Y))] \tag{3.18}$$

So it minimizes the loss i.e. the second term. And the KL divergence is acting as a regularizer to prevent overfitting.

### 3.6.6 Denoising Auto-Encoder

All of the previous subsections has been explained to get to this point: Denoising Auto-Encoder. Reminder the goal of the diffusion model is to denoise data ($Y = X + \sigma\varepsilon$) and find the noise $\varepsilon$, the denoising auto-encoder tries instead to find the original data $X$. We have those following structures:

- Auto-Encoder $D(E(X))$
- Variationnal Auto-Encoder $D(E(X) + \varepsilon)$
- Denoising Auto-Encoder $D(E(X + \sigma\varepsilon))$

To train the denoising auto-encoder, we can use the Evidence Lower-Bound with $Y = X + \sigma\varepsilon$ and $Z$ such that $X = D(E(Z))$:

$$-\log(f_X(x)) \leq D_{KL}((Y|X=x)\|Z) - \mathbb{E}[\log(f_{X|Z}(x|Y))] \tag{3.19}$$

## 3.7 Conditionned Diffusion Models

Conditioned diffusion generates data by denoising noise, but the denoising is guided by a conditioning signal (text, image, layout, etc.). Where unconditionned diffusion learn $p(x)$, conditionned diffusion learn $p(x|c)$, with $c$ the conditionning signal. It is for example used to train text-to-image generation, where the model generates an image based on a text description. This conditionned diffusion will need to learn the relationship between the text and the image so it will need cross-attention like in a transformers.

## 3.8 Classifier-Free Guidance

To improve the conditioned diffusion for multi-modal distributions, we can use a classifier but it need also training. So to overcome this, we can use classifier-free guidance. The idea is to train the model to handle both conditioned and unconditioned data. During training, we train both conditionned (with condition $\tau$) and unconditionned diffusion model and combine them with:

$$\bar{\epsilon}_t = \lambda\epsilon_\theta(x_t, \sigma_t, \tau) + (1-\lambda)\epsilon_\theta(x_t, \sigma_t) \qquad \lambda > 1 \tag{3.20}$$

## 3.9 Optical illusions

To generate optical illusions, we do not need to create a specialized model, we can use a pre-trained diffusion model. We can use this model and at each step, we denoise the image under $N$ transformations ($v_i$), bring all denoising directions back to the original space, average them, and use this average as the update. The optimization problem can be formulated as:

$$\bar{\epsilon}_t = \frac{1}{N} \sum_{i=1}^{N} v_i^{-1}(\epsilon_\theta(v_i(x_t), \sigma_t, \tau)) \tag{3.21}$$

The fact that we try to retrieve the same image under different transformations creates conflicting objectives that lead to the generation of optical illusions.

# Kernels

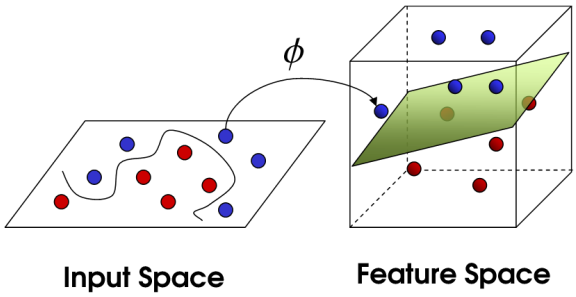



Figure 4.1: Illustration of kernels

A kernel is a function that transforms a dataset into another, typically of higher dimension. This helps separate the nonlinear feature, to use the usual linear tools. For example, the canonical kernel is

$$r(x,y) = (x^T y)^2 = \phi(x)^T \phi(y) \tag{4.1}$$

where the kernel function then is

$$\phi(x) = \begin{pmatrix} x_1^2 \\ x_2^2 \\ \sqrt{2}x_1 x_2 \end{pmatrix} \tag{4.2}$$

## 4.1 Reminders on scalar product

→ Reminder: a Euclidean space is a finite-dimensional vector space endowed with a scalar product.

A scalar product $\langle \cdot, \cdot \rangle_V$ verifies

- Symmetry: $\forall x, y \in V$, $\langle x, y \rangle_V = \langle y, x \rangle_V$;
- Definite positive: if $x \in V$ and $x \neq 0$, then $\langle x, x \rangle > 0$, and if $x = 0$, then $\langle x, x \rangle = 0$;
- Bilinearity: $\forall x, y, z \in V$, $\forall \alpha, \beta \in \mathbb{F}, \langle (\alpha x + \beta y), z \rangle_V = \alpha \langle x, z \rangle_V + \beta \langle y, z \rangle_V$.

### 4.1.1 Equivalence

Consider a positive definite symmetric matrix $M \succ 0 \in \mathbb{R}^{n\times n}$ with the scalar product $\langle x, y\rangle_M = x^T My$. Any such matrix can be factored as $M = L^T L$, where $L$ is invertible. We can do a change of coordinates to re-express under the cartesian scalar product:

$$\begin{cases} w = Lx \\ z = Ly \end{cases} \implies \langle x, y\rangle_M = \langle w, z\rangle_{\mathbb{R}^n} \tag{4.3}$$

Then, all $n$-dimensional Euclidean spaces are equivalent up to a change of coordinates.

### 4.1.2 Hilbert space

For functions integrable on an interval $[a, b]$, we define the scalar product as

$$\langle f, g\rangle = \int_a^b f(t)g(t)dt \tag{4.4}$$

## 4.2 Kernel methods for finite sets

### 4.2.1 Example – word embedding

Text embedding has as objective to represent a text with a vector. The general idea is one-hot encoding, i.e. for a data set $X$ of $N$ words, we create one-hot vectors of dimension $N$, using the feature map (or embedding) $\phi : X \to H$. A sentence is simply the sum of those vectors, weighted by the number of occurences. From this, we can create the kernel matrix $K$, where each element $K_{ij}$ is the comparison between the basis vectors $e_i, e_j$. We suppose that the $\phi$ function has a unitary norm.

$$K_{ij} = \langle \phi(x_i), \phi(x_j)\rangle_K \in [-1, 1] \tag{4.5}$$

A value of 1 means that the words $x_i$ and $x_j$ are identical, but if the value is $-1$, their meaning are opposite. In the case of a 0, there is no connection between the words. Now, we can calculate the Cholesky decomposition $K = L^T L$ for a new basis, and use the usual scalar product.

→ Note: if the matrix $K$ is not invertible, we just hit a degenerate case where the $\phi(x_i)$ are not linearly independent. This is not a problem.

### 4.2.2 Kernel trick

The kernel trick is to never explicitly define the function $\phi$: we can work with the matrix $K$ only, as any vector can be expressed as a linear combination of $\phi(x_i)$ and the main ingredient of linear methods is generally the scalar product, here defined only using $K$.

$$\langle \phi(y), \phi(z)\rangle = \langle \sum_{i=1}^N \alpha_i \phi(x_i), \sum_{i=1}^N \beta_i \phi(x_i)\rangle_K = \sum_{i=1}^N \sum_{j=1}^N \alpha_i \beta_j \langle \phi(x_i), \phi(x_j)\rangle_K = \alpha^T K \beta \tag{4.6}$$

### 4.2.3 Reconstruction

Given the matrix $K$, we can reconstruction the embedding ($\phi$ and $\langle\cdot,\cdot\rangle_K$) up to a rotation. For $X$ the set of words and $N$ the dimension of the space, we have $\phi : X \to \mathbb{R}^N$ and we define the scalar product:

$$\langle x, y\rangle_M = \langle x, y\rangle_{K^{-1}} \tag{4.7}$$

Then, remembering $\phi(x_i) = Ke_i$,

$$\langle \phi(x_i), \phi(x_j)\rangle_M = (Ke_i)^T K^{-1}(Ke_j) = e_i^T K^T K^{-1} K e_j = e_i^T K e_j \tag{4.8}$$

Let us prove the "up to a rotation" part. Define

$$\begin{cases} \phi' : X \to \mathbb{R}^N \\ \phi'(x) = Q\phi(x) \\ \langle x, y\rangle_{M'} = \langle x, y\rangle_{QK^{-1}Q^T} \end{cases} \tag{4.9}$$

where $Q$ is a rotation matrix, i.e. $Q^T Q = I$. Then,

$$\begin{aligned} \langle \phi'(x), \phi'(y)\rangle_{M'} &= (Q\phi(x))^T QK^{-1}Q^T(Q\phi(y)) \\ &= \phi(x)^T K^{-1}\phi(y) = \langle \phi(x), \phi(y)\rangle_{K^{-1}} \end{aligned} \tag{4.10}$$

The result is true for any rotation matrix $Q$.

→ Note: the complexity of computing $K$ is $\mathcal{O}(N^2)$, while the Cholesky decomposition to find the new basis has a complexity of $\mathcal{O}(N^3)$. This decomposition is to be avoided if not really necessary.

### 4.2.4 Advantages

Let us consider proteins. The alphabet is of size 20 (number of existing amino-acids), and a protein is made of 4 of these. This means that our space is initially $H = \mathbb{R}^{20^4} = \mathbb{R}^{160.000}$. Consider $N = 100$ proteins to classify.
The embedding consists in storing $N$ vectors of size $h = 160.000$, meaning 1.6M scalars. However, we only need the $K$ matrix of size $N^2 = 10.000$, and the entries are easy to compute. This means that, using the kernel trick, i.e. working with $K$ directly, we are much more efficient when $N << h$.

## 4.3 Kernels methods for continuous sets

### 4.3.1 Reproducibility Kernel Hilbert Space

Consider richer sets than finite $X$, e.g. infinite or uncountable sets, with distances defined but not scalar product. Then, let $H$ be a vector space of conitnuous functions $X \to \mathbb{R}$. $H$ is a RKHS if it is a Hilbert space and

$$\forall x \in X, \exists v \in H \text{ such that } \forall f \in H : \langle v, f\rangle_H = f(x) \tag{4.11}$$

Mathematically, the kernel function is the equivalent of the kernel matrix, but for infinite spaces:

$$\phi : X \to \mathbb{R}^N \text{ such that } k(x, y) = \langle \phi(x), \phi(y)\rangle_K = \alpha_x^T K \alpha_y \tag{4.12}$$

For a RKHS $H$, we verify

$$\begin{cases} \forall x \in X : k(x,\cdot) \in H \\ \forall x \in X, \forall f \in H, f(x) = \langle f, k(x,\cdot)\rangle_K \end{cases} \tag{4.13}$$

The reproducing kernel property is

$$k(x,y) = \langle k(x,\cdot), k(y,\cdot)\rangle_H \tag{4.14}$$

### 4.3.2 Properties

**Theorem 4.1.** A continuous map $k : X \times X \to \mathbb{R}$ is the kernel of some RKHS $H \subset \mathscr{C}(X,\mathbb{R})$

- iff $k(x,y) = \langle \phi(x), \phi(y)\rangle_H$ for some feature map $\phi : X \to H$;
- iff, for all finite subsets $X_0 \subset X$, the kernel matrix is symmetric positive definite.

## 4.4 Polynomial kernels

Polynomial kernels are a particular type of continuous kernels, where the feature map is a function

$$\phi : R^n \to \mathbb{R}^\ell \tag{4.15}$$

Let us take our initial example 4.2, and generalize it to

$$k(x,y) = (x^T y)^d \qquad X = \mathbb{R}^n \tag{4.16}$$

Then, the dimension of the out space $H = \mathbb{R}^\ell$ is $\ell = \binom{n+d-1}{d}$.