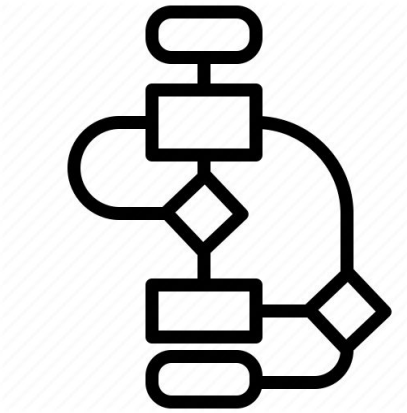


LINMA2472 - Algorithm in data science

Alexandre Orékhoff
Issambre L'Hermite Dumont
Simon Desmidt

Academic year 2025-2026 - Q1

UCLouvain

Contents

1 Automatic Differentiation **3**
- 1.1 Chain rule . . . 3
- 1.2 Forward differentiation . . . 3
- 1.3 Backward differentiation . . . 4
- 1.4 Computational graph and multivariate differentiation . . . 5
- 1.5 Jacobian computation . . . 6
- 1.6 Memory usage . . . 7
- 1.7 Second order AD . . . 8
- 1.8 Implicit differentiation . . . 15
- 1.9 Sparse AD . . . 18

2 Neural networks **24**
- 2.1 Autoregressive models . . . 25
- 2.2 Tokenization . . . 26
- 2.3 Embedding . . . 27
- 2.4 Recurrent neural networks (RNN) . . . 28
- 2.5 Attention is all you need . . . 29
- 2.6 Decoder-only transformer . . . 31
- 2.7 Performances of transformers . . . 34

3 Diffusion Models **35**
- 3.1 Tweedie's formula . . . 35
- 3.2 Langevin dynamics (sampling) . . . 35
- 3.3 Score matching . . . 36
- 3.4 Deterministic Sampling . . . 36
- 3.5 Denoising with randomness . . . 37
- 3.6 Auto-Encoder . . . 37
- 3.7 Conditioned Diffusion Models . . . 41
- 3.8 Classifier-Free Guidance . . . 41
- 3.9 Optical illusions . . . 41

4 Kernels **42**
- 4.1 Reminders on scalar product . . . 42
- 4.2 Kernel methods for finite sets . . . 43
- 4.3 Kernels methods for continuous sets . . . 44
- 4.4 Polynomial kernels . . . 45
- 4.5 Operations on kernel . . . 45
- 4.6 Gaussian kernel . . . 46
- 4.7 Random Fourier Features . . . 46

4.8 Applications of kernels 48

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.

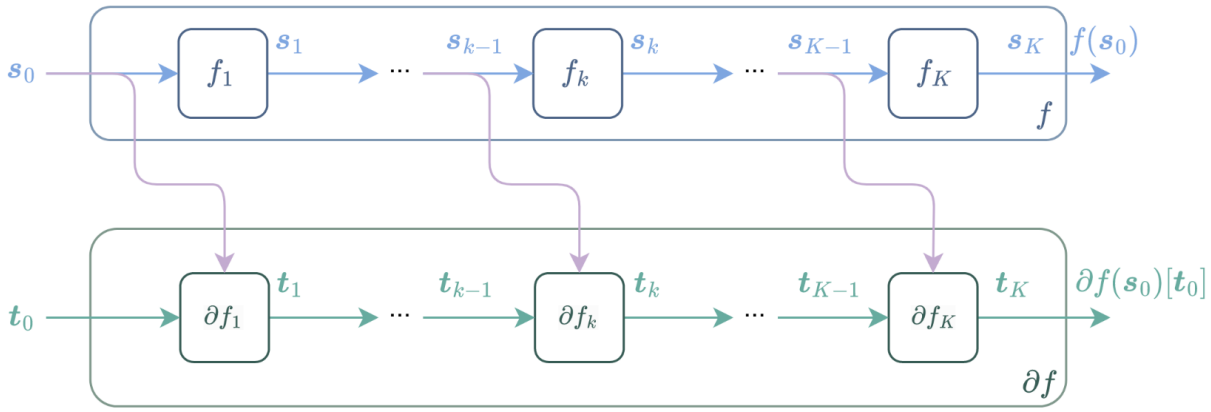


Figure 1.1: Forward differentiation

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.

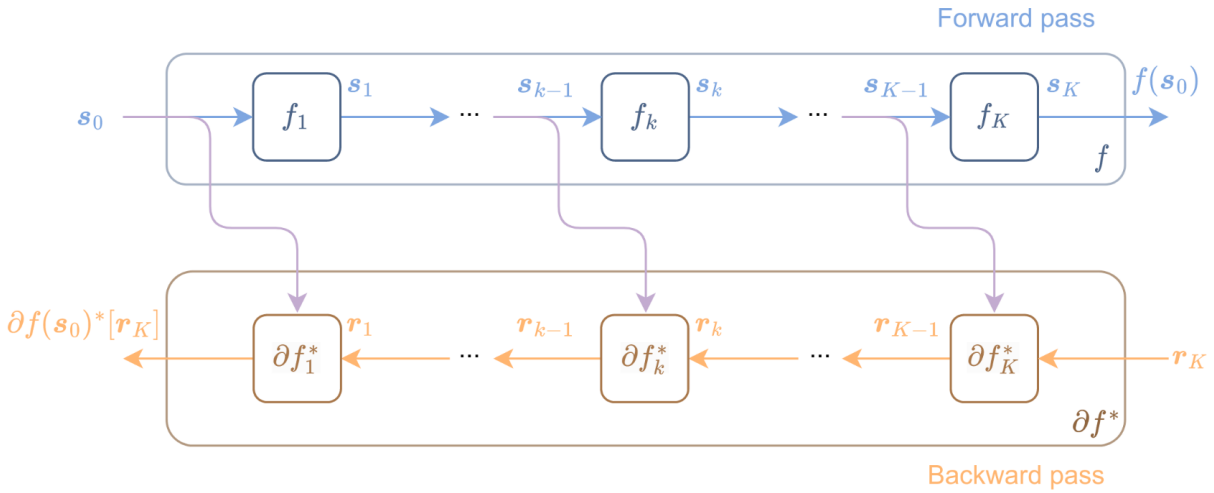


Figure 1.2: Backward differentiation

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:

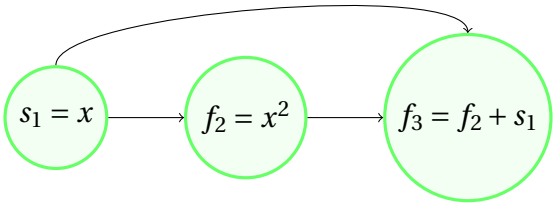


Figure 1.3: Directed Acyclic Graph (DAG) of f_3

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 prevails.

1.6 Memory usage

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

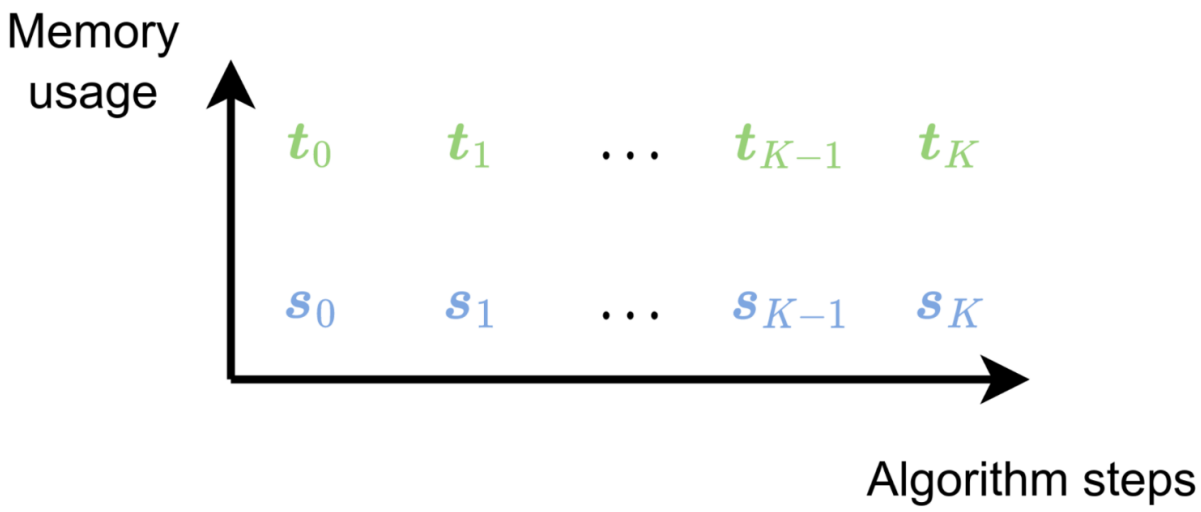


Figure 1.4: Forward mode memory usage

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.

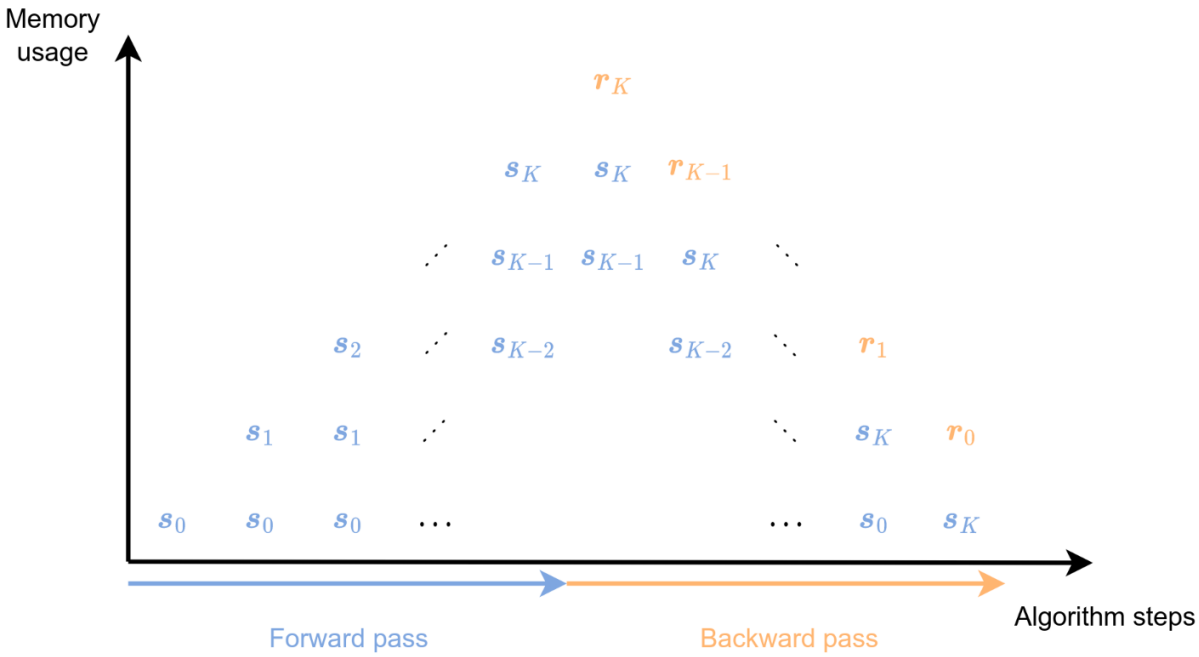


Figure 1.5: Backward mode memory usage

Concluding with this section, the forward mode is deemed more memory efficient than the backward mode. However, this aspect is, in general, 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 dimension $n \times n$. Its entries can be computed as

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

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 both be 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 ε 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.

$$f(a + b\varepsilon) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)\, b^n \varepsilon^n}{n!} \overset{\varepsilon^{\geq 2}=0}{=} f(a) + bf'(a)\varepsilon \tag{1.20}$$

Back to the automatic differentiation concept, $f(a)$ is the function 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 encoding 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, and 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 the primal values;
2. a backward pass computing the adjoints of the primal values;
3. a forward pass computing the tangents the adjoint values.

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

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.

```
function FoR(f, x, v):

    # 1. Forward pass (primal only)
    tape = []
    y = forward_primal(f, x, tape)
    # tape stores all ops and intermediate primal values

    # 2. Reverse pass (gradient)
    bar = zero_like(tape.primal_vars)
    bar[y] = 1.0

    for op in reverse(tape):
        reverse_adjoint(op, bar)

    grad = bar[x]                    # grad f(x)

    # 3. Forward pass over reverse
    dot = zero_like(bar)
    dot[x] = v                       # seed tangent

    for op in reverse(tape):
        forward_tangent_of_reverse(op, bar, dot)
```

```
    Hv = dot[x]                    # Hessian vector product

    return grad, Hv
```

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 reads

- a forward pass propagating the primal values and their tangent;
- a backward pass computing the adjoints of both primal and tangent values;

Defining $s_1 \triangleq \text{Dual}\left(f_1(x) \;\middle|\; \frac{\partial f_1}{\partial x_i}\right)$ [3], the algorithm follows

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

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

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

Pseudo-code

```
function RoF(f, x, v, w):

    # 1. Forward pass (primal + tangent)
    tape = []
    dot = {}
    dot[x] = v                          # seed tangent

    y = forward_primal_tangent(f, x, dot, tape)

    # y      : f(x)
    # dot[y] : J(x) v

    # 2. Reverse pass
        # bar accumulates d/d(primal)
        # bar_dot accumulates d/d(tangent)
    bar = zero_like(tape.primal_vars)
    bar_dot = zero_like(tape.tangent_vars)

    bar_dot[y] = w                      # seed cotangent

    for op in reverse(tape):
        reverse_over_forward(op, bar, bar_dot)

    grad = bar[x]                       # grad <Jv, w>

    return grad
```

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-reverse 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

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

During the forward pass, the K intermediary values are stored, like in the 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, $\nabla f(x) = r_0$ is obtained. All adjoints are stored, as they are required for the second reverse pass.

Second reverse pass (second-order adjoints)

The adjoint equations are differentiated 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 $\dfrac{\partial s_{k-1}}{\partial x_i}$.

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 Hessian 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 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 depend 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 ω 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))**.** Assume

- $F : \mathcal{W} \times \Lambda \to \mathcal{W}$ (w is the variable and λ is a parameter);
- $\exists (w_0, \lambda_0)$ 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, λ_0).

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

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

In the example of the square root above, w corresponds to a and λ to x (notations have been inversed, ask BenoÃőt).

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.

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

Let us consider that case where the function that we want to differentiate is the composition of explicit and implicit functions. This can happen in numerical methods such as the gradient descent. The implicit function is only one step in a chain rule, meaning that we can put it in the forward tangent recurrence relation.

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

$$t_k = -\left(\partial_1 F(w^*, \lambda)\right)^{-1} \cdot \partial_2 F(w^*, \lambda) \cdot t_{k-1} \tag{1.44}$$

where, for that specific step, $t_k = \partial w^*(\lambda)/\partial s_0$ and $t_{k-1} = \partial \lambda / \partial s_0$. Each JVP of the implicit function may be obtained 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:

$$t_k = A^{-1} B t_{k-1} \tag{1.45}$$

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.

$$\begin{aligned} LU t_k &= B t_{k-1} \\ &\Downarrow \\ \begin{cases} Ly &= B t_{k-1} \\ U t_k &= y \end{cases} \end{aligned} \tag{1.46}$$

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.

$$r_{k-1} = -\left(\partial_2 F(w^*, \lambda)\right)^T \cdot \left(\partial_1 F(w^*, \lambda)\right)^{-T} \cdot r_k \tag{1.47}$$

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.

$$\begin{cases} A^T y = r_k \\ r_{k-1} = -B^T y \end{cases} \tag{1.48}$$

1.8.2 AD with optimization problem

Consider the optimization problem defined by:

$$\min c^T x \quad \text{s.t.} \quad Ax = b, \quad x \geq 0 \tag{1.49}$$

and its dual:

$$\max b^T y \quad \text{s.t.} \quad A^T y \leq c \tag{1.50}$$

The KKT conditions give the optimality conditions for this problem:

$$\begin{cases} Ax = b \\ \left[A^T y - c\right] \perp [x \geq 0] \end{cases} \tag{1.51}$$

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

$$F((x,y),(A,b,c)) = \begin{bmatrix} Ax - b \\ Diag(x)(A^T y - c) \end{bmatrix} \tag{1.52}$$

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

$$\frac{\partial F}{\partial (x,y)} = \begin{bmatrix} A & 0 \\ Diag(A^T y - c) & Diag(x) A^T \end{bmatrix} \tag{1.53}$$

$\partial_2 F(w^*, \lambda)$ can also be computed, and we can use the IFT to get the final sensitivity of the problem when the parameters change.

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:

$$J = \begin{bmatrix} 0 & * & 0 & * & 0 \\ 0 & 0 & 0 & * & * \\ 0 & * & * & 0 & 0 \\ * & 0 & * & 0 & 0 \end{bmatrix} \tag{1.54}$$

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 computing 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

$$J \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ * \end{bmatrix} \qquad \text{and} \qquad J \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} * \\ 0 \\ * \\ 0 \end{bmatrix} \tag{1.55}$$

Combining these two JVPs gives:

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

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:

$$\begin{bmatrix} 0 & * & 0 & * & 0 \\ 0 & 0 & 0 & * & * \\ 0 & * & * & 0 & 0 \\ * & 0 & * & 0 & 0 \end{bmatrix} \quad \Rightarrow \quad v_1 = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad v_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{1.57}$$

Or for the VJP:

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

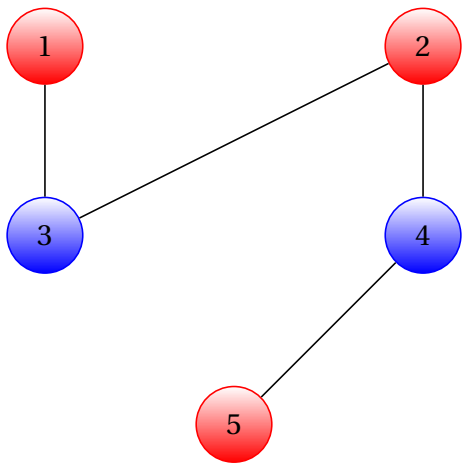
1.9.1 Sparsity detection

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:

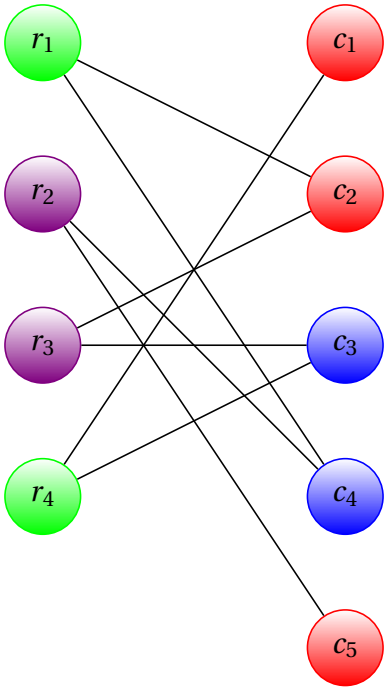
$$J = \begin{bmatrix} 0 & * & 0 & * & 0 \\ 0 & 0 & 0 & * & * \\ 0 & * & * & 0 & 0 \\ * & 0 & * & 0 & 0 \end{bmatrix} \tag{1.59}$$



→ Note: in more complex sparsity patterns, somes nodes could be linked multiple times.

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

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



With this modeling, the coloring problem is a distance-2 graph coloring problem. Two column nodes being at distance 2 from each other cannot 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.

$$J = \begin{bmatrix} a_1 & a_2 & a_3 & 0 \\ a_2 & a_4 & 0 & a_5 \\ a_3 & 0 & a_6 & 0 \\ 0 & a_5 & 0 & a_7 \end{bmatrix} \tag{1.61}$$

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

$$J = \begin{bmatrix} a_1 & a_2 & a_3 & 0 \\ a_2 & a_4 & 0 & a_5 \\ a_3 & 0 & a_6 & 0 \\ 0 & a_5 & 0 & a_7 \end{bmatrix} \tag{1.62}$$

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

Computing the two JVPs and retrieving the full Jacobian matrix shows

$$J\begin{bmatrix}1 & 0\\ 0 & 1\\ 0 & 1\\ 1 & 0\end{bmatrix} = \begin{bmatrix}a_1 & a_2+a_3\\ a_2+a_5 & a_4\\ a_3 & a_6\\ a_7 & a_5\end{bmatrix} \tag{1.63}$$

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 VJPs by grouping rows instead of columns.

→ Note: as the hessian matrix is symmetric, this property is very useful for sparse second-order AD.

1.9.3 Star coloring

Given a graph $G = (V, E)$, and a coloring function $\phi : V \to \{1, \dots, p\}$, 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

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

Coloring its associated graph with the classic distance-1 coloring would yield the same matrix as (1.62). Computing the JVPs, one would obtain the matrix (1.63) 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

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

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

$$J\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix} = \begin{bmatrix}a_1 & a_2+a_3 & 0\\ a_2 & a_4 & a_5\\ a_3 & a_6 & 0\\ 0 & a_5 & a_7\end{bmatrix} \tag{1.66}$$

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 JVPs), and the symmetric distance-1 coloring (2 JVPs + linear system).

1.9.4 Acyclic coloring

For any Hessian H and vector v, computing

$$y = Hv$$

gives a vector y whose components are defined as

$$y_i = \sum_{j \in \mathcal{N}(i)} H_{ij} v_j \tag{1.67}$$

where $\mathcal{N}(i)$ is the set of non-zero entry column indexes at row i.

From here, assume $\phi : V \to \{1, \ldots, p\}$ is a proper coloring such that no two neighboring nodes have the same color.

Diagonal entries

Diagonal values are easy to extract. Assume node i is colored in c. Then given the vector $v^{(c)}$ whose j-th component is 1 iff j is colored in c, the i-th component of the HVP yields:

$$(Hv^{(c)})_i = H_{ii}$$

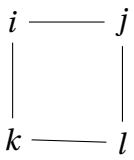
Off-diagonal entries

It is readily seen that for a node i, colored c_1 and linked to a node j, colored c_2, $H_{ij} = H_{ji}$ appears in $(Hv^{(c_2)})_i$ and $(Hv^{(c_1)})_j$. Unfortunately, due to eq.1.67, these entries appear summed with others. For instance

$$(Hv^{(c_2)})_i = H_{ij} + H_{ik} + H_{il}$$

Cycles

Cycles are a problem in coloring. Assume column nodes i, j, k, l are connected in a cycle.



Following eq.1.67, equations for those components would be

$$\begin{cases} y_i = H_{ij} + H_{ik} \\ y_j = H_{ji} + H_{jl} \\ y_k = H_{ki} + H_{kl} \\ y_l = H_{lj} + H_{lk} \end{cases}$$

which are cycling equations. Solving this system requires additional HVPs.

Acyclic coloring

An *acyclic coloring* ensures that for any two colors c_1, c_2, the induced subgraph has no cycle. It is, therefore, a *tree* or a *forest* where every connected component has at least one leaf.

The algorithm follows

1. Suppose node i (colored c_1) is connected to exactly one node j (colored c_2): i is a leaf.
2. From the HVP from color c_2, one obtains

$$(Hv^{(c_2)})_i = H_{ij}$$

   Indeed, since i is a leaf of j, and thus the link $(i-j)$ is the only edge i has, the only contribution to the i-th component of the j-th HVP (see eq. 1.67) is H_{ij}.
3. Once H_{ij} is known, it can be substracted from the equation at j, the node i can be removed from the tree, and the tree shrinks.
4. The process is repeated until the full Hessian is constructed.

This strategy only requires as much HVPs as there are colors required to form an acycling coloring, and often

$$\#\text{colors} \ll \#\text{variables}$$

Mentally, this process can be seen as

- Each HVP gives blurred information;
- Acyclic coloring makes sure that blur can always be unblurred locally.

1.9.5 Chromatic number (ξ)

For every graph G,

$$\xi_1(G) \leq \xi_{acyclic}(G) \leq \xi_{star}(G) \leq \xi_2(G) \tag{1.68}$$

ξ_1 is the usual distance-1 chromatic number, ξ_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: weight matrix at layer k;
- b_k: bias vector at layer k;
- σ: activation function (ReLU, sigmoid, etc);
- $\ell(.)$: loss function;
- H: number of hidden layers;
- S_i: intermediate state.

A neural network can be seen as a composition of functions, the type of which was discussed in sections above. A neural network needs to be trained, or in other words its parameters have to be optimized. Optimization requires computing derivatives and that's where automatic differentiation comes to play.

The forward pass of a neural network can be described 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 main difference resides in the fact that weights are applied on the left side or on the right side of the data. It can be useful depending on whether the inputs are represented as rowvectors or columnvectors.

Figures 2.1 and 2.2 shows the respective computational graphs.

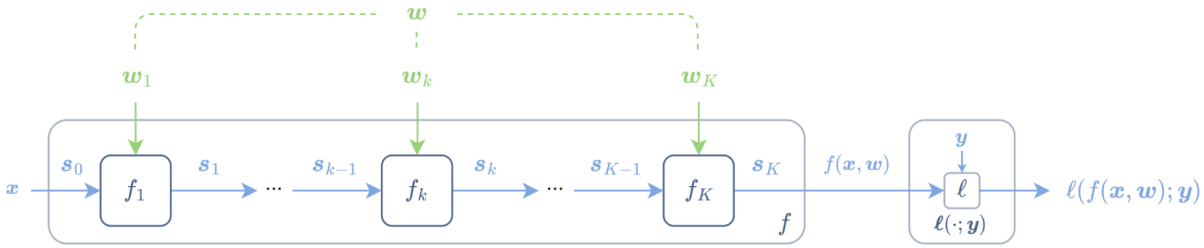


Figure 2.1: Neural network forward pass

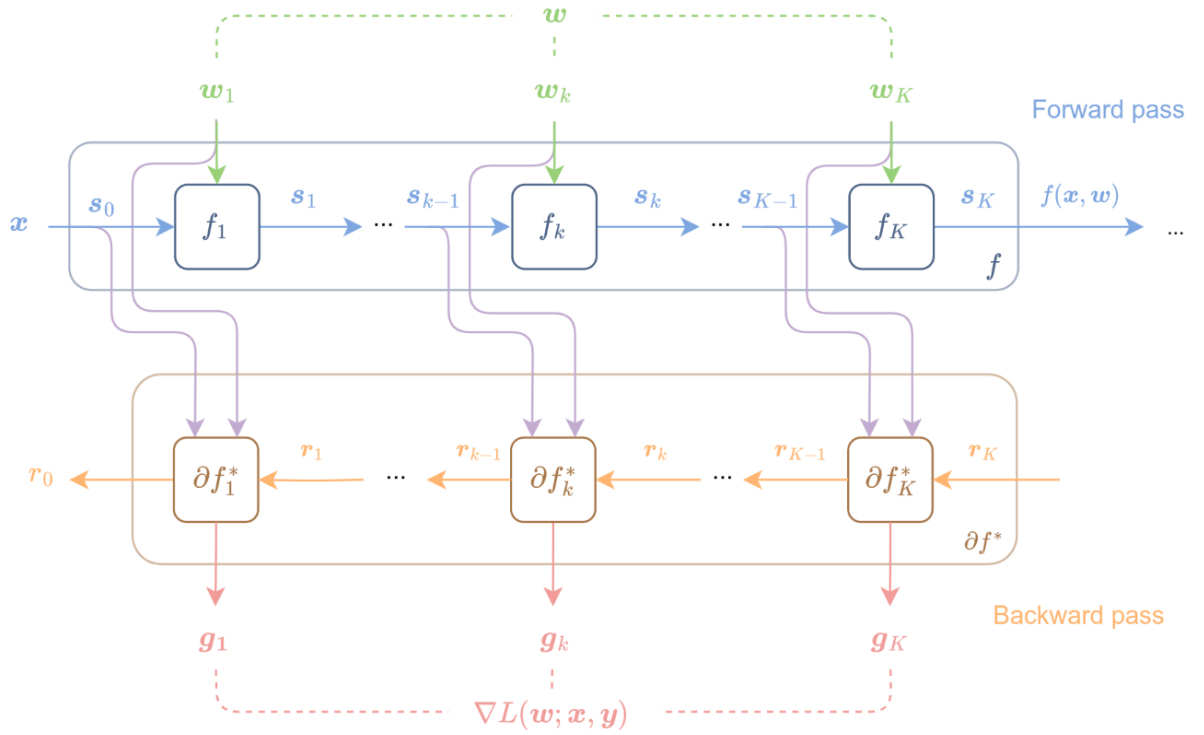


Figure 2.2: Neural network backward pass

Following sections will focus on the building blocks leading to a *transformer*, one type of neural network, involved in Natural Language Processing (NLP) and extensively used nowadays in models such as *ChatGPT*, *Claude*, or *LLama*.

The first step consists in defining the mathematical model. Once it is obtained, one shall find a strategy to encode text into vectors efficiently. Eventually, the full transformer architecture may be presented.

2.1 Autoregressive models

Autoregressive models want to predict the next values in a sequence based on previous values.

Given a sequence of n_{ctx} (context size) past vectors $x_{-1}, x_{-2}, \ldots, x_{-n_{ctx}}$, named *receding horizon*, the model aims to predict the next vector x_0 and maybe more $(x_1, x_2, \ldots)$.

For instance, one may want to predict x_0 and x_1 based on past data.

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

- **Model**: Probability of next vector $\hat{p}(x_0 \mid X)$ where X concatenates $x_{-1}, \dots, x_{-n_{ctx}}$
- **Loss**: Cross-entropy to assess the error of $\hat{p}$ w.r.t. p

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

In the particular (and common) case where the probability distribution p puts all the mass on the expected output vector y, $p(x_0 \mid X) = \delta_y(x_0)$ where $\delta_y(x_0) = 1$ if $x_0 = y$ and 0 otherwise.

Accordingly, the cross-entropy can be simplified

$$\begin{aligned} \mathscr{L}(X) &= -\sum_{x_0} \delta_y \log(\hat{p}(x_0|X)) \\ &= -\log(\hat{p}(y|X)) \end{aligned}$$

2.2 Tokenization

> First step of word vectorization is *tokenization*.

Consider one has n_{ctx} characters of context and wants to predict the next characters. The one-hot encoding would lie in $\mathbb{R}^{26}$, thus $n_{voc} = 26$. It imposes n_{ctx} to be very large, which is not a good news because transormers have a quadratic complexity in n_{ctx}.

Another strategy consists in one-hot-encoding each word, but the vocabulary size is often very large (+200k words). While this strategy could lead to relative low n_{ctx}, n_{voc} would be too big.

2.2.1 Byte Pair Encoding (BPE)

An intermediate solution is to use a *Byte Pair Encoding* algorithm, which greedily merges the most frequent pairs of tokens into new tokens.

For instance, considering the word `abracadabra` at the character level, one may observe that it has two frequent pairs `ab` and `ra`, that can be merged into new tokens X and Y respectively.

$$\texttt{abracadabra} \rightarrow \texttt{XYcadXY} \tag{2.3}$$

This process can be repeated until the desired vocabulary size is reached.

The next iteration could give:

$$\texttt{XYcadXY} \rightarrow \texttt{ZcadZ} \tag{2.4}$$

This tokenization method allows to have a good trade-off between the vocabulary size n_{voc} and the context size n_{ctx}.

However, if the model is not trained enough, it could return a sequence of tokens making no sense (a sequence of words not related to the previous token) if this sequence is the most common token present after the given input context (for instance, returning "is my daughter" after "During the month of April") in the training corpus.

2.2.2 Byte BPE

Byte BPE (Byte Byte Pair Encoding) consists in replacing every character by its numerical value in an ASCII extended table (for instance the ISO 8859-1 table). This allows the model to be more flexible on word encodings and accept any language.

The greatest strength of the Byte BPE encoding consists in separating radicals from prefixes and suffixes by working at character level. For instance, the words **optimize**, **optimiz**ing, **opimiz**ation, ..., all have the radical **optimiz** in common and the Byte BPE encoder would be able to determine that **optimiz** is an enough repeating sequence of character to form a single token. Instead of storing the 3 words separately, it will store the radical **optimiz**, and the suffixes *e*, *ing* and *ation*, which are pretty common in other English words.

However, if the encoder is not trained enough, it could be affected by the same issue affecting the usual BPE described above.

2.3 Embedding

> Second step of word vectorization is *embedding*.

Consider a vocabulary of size n_{voc}, a bigram model and a network with d layers.

The model would be:

$$\hat{p}(x_0|x_{-1}) = \text{softmax}(W_d \tanh(\dots \tanh(W_1 x_{-1}) \dots)) \tag{2.5}$$

The matrix W_1 has n_{voc} columns and W_d has n_{voc} rows. When n_{voc} is large, the model becomes problematically big.

The main idea resides in using a pair of encoder/decoder to reduce the size. This technique is called *embedding* and it reduces the vocabulary size to the embedding size d_{emb} with $d_{emb} \ll n_{voc}$. 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}.

The model becomes

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

Choosing d_{emb} wisely ($d_{emb} \ll n_{voc}$), it becomes faster to compute $W_1(Cx_{-1})$ in the new embedding size. Moreover, this forces $W_1 C$ to be low-rank which can help reduce overfitting but it reduces the *expressiveness* (capacity to capture a range of possible relation between input and output) of the model. When $n_{ctx} > 1$, it is possible to compute the embedding matrix once for every vector in the input context.

In experimentation, it appeared that forcing $D = C^T$ works well in practice.

2.3.1 Shared embedding

When $n_{ctx} > 1$, the encoder C is shared by all tokens, and the model can be written as

$$\hat{p}(x_0|x_{-1}, \dots, x_{-n_{ctx}}) = \text{softmax}(DW_d \tanh(\dots \tanh(W_1 \begin{bmatrix} Cx_{-1} \\ \vdots \\ Cx_{-n_{ctx}} \end{bmatrix}) \dots)) \tag{2.7}$$

W_1 is a matrix of dimension $n_{voc} \times n_{ctx} d_{emb}$. Assuming $d_{emb} \ll n_{voc}$ and $n_{ctx} \gg 1$, this is much smaller than $n_{ctx} n_{voc}$ occurring before the embedding phase. The number of rows of W_2 is not affected by the embedding and remains n_{voc}.

Figure 2.3 depicts the model with shared embedding.

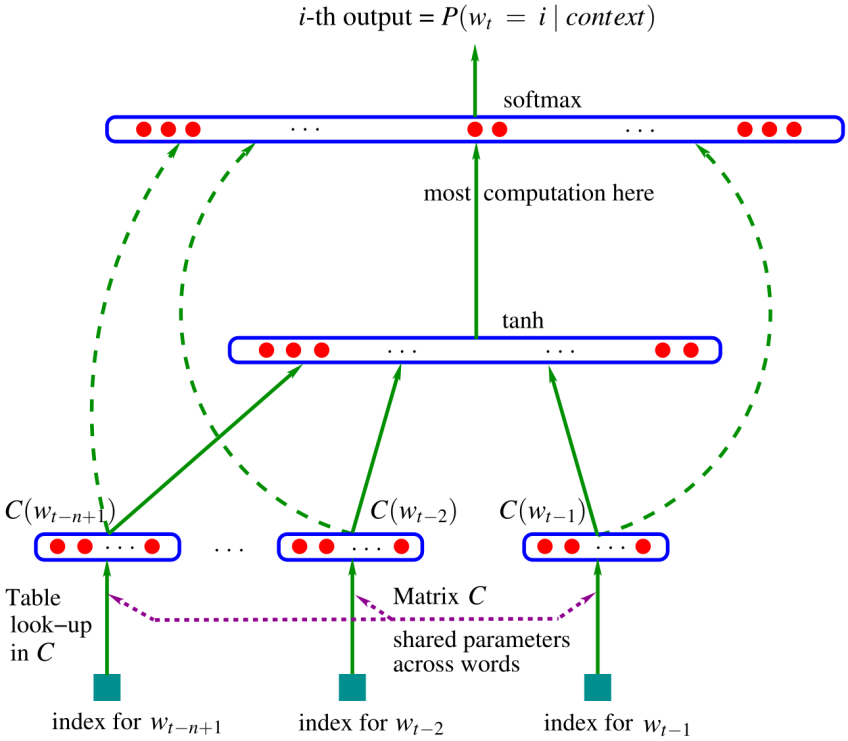


Figure 2.3: Neural network with shared embedding

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:

$$\begin{cases} h_{t+1} = \tanh(W h_t + U x_{t+1} + b) \\ \hat{y}_t = \text{softmax}(V h_t + c) \end{cases} \tag{2.8}$$

where W, U, V are weight matrices and b, c are bias vectors.

This model presents some limitations:

- Long-term dependencies hard to learn due to vanishing/exploding gradient problem;
- Hard to parallelize due to step-by-step sequence processing;
- Memory-intensive for long sequences due to hidden state storage at each time step.

2.5 Attention is all you need

2.5.1 Attention

Numerical Dictionary

Consider keys $k_i \in \mathbb{R}^{d_k}$, values $v_i \in \mathbb{R}^{d_v}$. Given a query $q \in \mathbb{R}^{d_k}$, one may want to retrieve the value v_i corresponding to the key k_i that is the most similar to the query q. The adapted tool for this task is the dot-product.

For each token, the query q_i, the key k_i, and the vaue v_i may be seen as answers to the following questions:

- q_i: What am I looking for?
- k_i: What do I contain?
- v_i: What information do I pass if I am selected?

One key aspect of this numerical dictionary is its *differentiability*. This property will be useful when training the full model.

Attention definition

With a numerical dictionary defined as above, *attention head* is defined as

$$\text{Attention}(Q,K,V) = \sum_{i=1}^{n_{ctx}} \alpha_i v_i \tag{2.9}$$

where $\alpha = \text{softmax}(\langle q,k_1\rangle, \cdots, \langle q,k_{n_{ctx}}\rangle)$ is the attention weight for key k_i. In practice, however, queries, keys and values are vectors. They are respectively contained in matrices Q,K,V.

With the matrix definition, the *attention head* may be rewritten.

$$\text{Attention}(Q,K,V) = V\text{softmax}\left(\frac{K^TQ}{\sqrt{d_k}}\right) \tag{2.10}$$

where the division by $\sqrt{d_k}$ is used to prevent the dot-product from growing too large.

Figure 2.4 shows the heatmap of the matrix

$$K^TQ = \begin{bmatrix} k_1 & \dots & k_{n_{ctx}} \end{bmatrix} \begin{bmatrix} q_1 \\ \vdots \\ q_{n_{ctx}} \end{bmatrix} = \begin{bmatrix} \langle k_1,q_1\rangle & \dots & \langle k_1,q_{n_{ctx}}\rangle \\ \vdots & \ddots & \vdots \\ \langle k_{n_{ctx}},q_1\rangle & \dots & \langle k_{n_{ctx}},q_{n_{ctx}}\rangle \end{bmatrix}$$

This heatmap provides intuition for the attention mechanism. It is the key characteristic that allows the transformer to capture the meaning of a sentence by combining the meaning of individual words.

In particular, it helps the model understand that *blue* and *monster* are not two independant concepts, but rather form a single idea : *a blue monster*.

The heatmap shown represents the attention weights of a transformer trained for translation. It clearly shows that although *européenne* and *European* do not appear at the same position in the source and target sentences, the transformer is able to determinate that they share a closely related meaning.

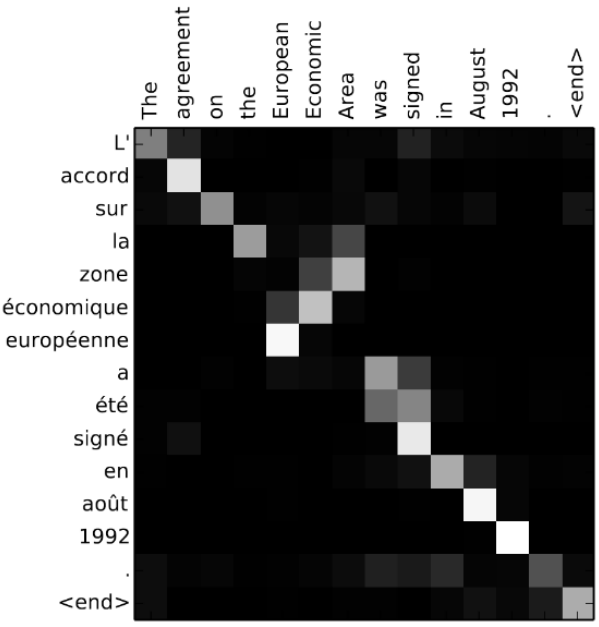


Figure 2.4: Heatmap representation of the attention of a transformer trained for translation.

2.5.2 Masked attention

To ensure that the model only attends to previous tokens and not future tokens, a masked attention mechanism must be used. 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.

With masked attention, the equation becomes

$$M = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ -\infty & 0 & \ddots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ -\infty & -\infty & \cdots & 0 \end{bmatrix} \tag{2.11}$$

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

2.5.3 Multi-head attention

Instead of having a single attention head, having multiple attention heads can help the model understand different aspects of the input data. Each head has its own set of weights and computes its own attention output.

After computing all the heads in parallel, their outputs are concatenated and linearly transformed to fit the dimension of W^O, the output weights matrix. This corresponds to a gathering of all of the meanings of the attention different heads to a single representation.

The multi-head attention mechanism can be defined as

$$\begin{cases} \text{head}_j = \text{Attention}(W_j^Q Q, W_j^K K, W_j^V V) \\ \\ \text{MultiHead}(Q,K,V) = W^O \begin{bmatrix} \text{head}_1 \\ \text{head}_2 \\ \vdots \\ \text{head}_h \end{bmatrix} \end{cases} \tag{2.12}$$

2.6 Decoder-only transformer

Decoder-only is a transformer architecture used with autoregressive models, predicting the next token based on the past known sequence. In this architecture, matrices Q, K, V are the same, and are defined as the input tokens CX, where C is the embedding matrix. Hence, a decoder-only transformer computes $\text{MultiHead}(CX, CX, CX)$.

Figure 2.5 depicts the decoder-only transformer architecture that will be discussed in the following sections. The key component of it, attention, has already been detailed in previous sections.

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.

One-hot encoding cannot be used for the position because the dimension would be n_{ctx} and not d_{emb} as expected. The classic approach uses sines and cosines with an angle depending on the position of the token and the embedding dimension.

The positional encoding of a token at position pos, with the embedding index i, and embedding dimension d_{emb}, could be defined as

$$\begin{cases} \text{angle} = \dfrac{\text{pos}}{10000^{2i/d_{emb}}} \\ \text{PE}[\text{pos}, 2i] = \sin\big(\text{angle}\big) \\ \text{PE}[\text{pos}, 2i+1] = \cos\big(\text{angle}\big) \end{cases} \tag{2.13}$$

The positional encoding is added to the input embedding before applying the attention mechanism

$$\text{MultiHead}(CX + PE, CX + PE, CX + PE) \tag{2.14}$$

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 (see next section). [1]

This mitigates the vanishing gradient problem and helps the network's learning by reminding it of the original input.

[1] It corresponds to the *Add* part of a *Add&Norm* layer.

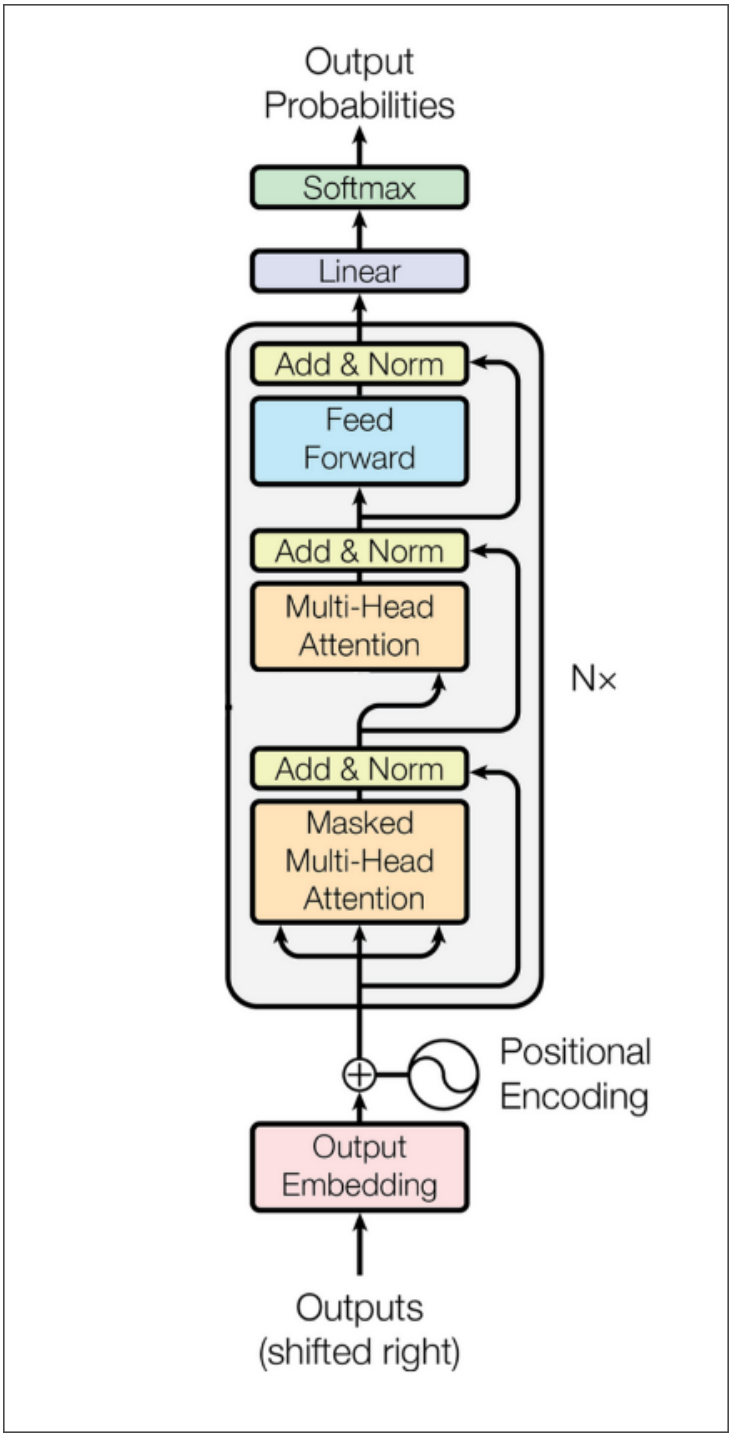


Figure 2.5: Decoder-only transformer architecture

2.6.3 Layer normalization

The norm of the gradient increases exponentially with the number of layers. Multiple techniques have been developped to remediate to this problem.

Two normalization techniques are typically considered.

1. Batch normalization computes the mean and variance per **feature**, across all *samples*. It aims to normalize the importance of a feature for every sample. For a feature j, every sample will have the same mean and variance.

   Due to its dependence on the number of samples, this technique performs poorly for RNNs and autoregressive models, which are based on sequences.

2. Layer normalization computes the mean and variance per **sample**, across all *features*. It aims to normalize the importance of all features in a sample such that none is preponderant.

   Since it does not depend on the number of samples, layer normalization is usually preferred for transformers.

A *normalization layer* helps stabilize the training process and improve convergence by rescaling the data before the activation function. It helps keep data's mean and variance under control.

It can be formulated as

$$\text{LayerNorm}(y_i) = \gamma \frac{y_i - \mu_i}{\sigma_i} + \beta \tag{2.15}$$

where γ and β are two learnable parameters, μ_i the mean of y_i's features, and σ_i their standard deviation.

In the transformer architecture, the *Layer Norm* always follows another layer needing normalization.

$$\text{LayerNorm}(\text{MultiHead}(CX + PE, CX + PE, CX + PE) + (CX + PE)) \tag{2.16}$$

2.6.4 Feed-forward network (FFN)

A *feed-forward layer*, also known as *dense layer*, simply[2] consists in a matrix product, a vector sum, and an activation function (ReLU[3]), applied to each token independently and identically,

$$\text{FFN}(x) = W_2 \text{ReLU}(W_1 x + b_1) + b_2 \tag{2.17}$$

with

- $W_1 \in \mathbb{R}^{d_{ff} \times d_{emb}}$;
- $W_2 \in \mathbb{R}^{d_{emb} \times d_{ff}}$;
- d_{ff} the dimension of the feed-forward layer. Empirically, it appears that $d_{ff} = 4 d_{emb}$ works nicely.

[2] lovely

[3] ReLU, for Rectified Linear Unit, is defined as $\text{ReLU}(x) = max(0, x)$

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 to remember more complex information in the weights[4], which helps the model better understand the complex relationships between tokens in the sequence.

2.6.5 Transformers variation

To investigate ?

2.7 Performances of transformers

Before analyzing the complexity of a transformer, here is a quick reminder of the key dimensions used in a transformer:

- 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;

The time complexity of a transformer can be analyzed step by step, ignoring the embedding step:

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

This brings the total cost for a single layer to:

$$\mathcal{O}\left((d_k + d_v) n_{ctx}^2 + (d_k + d_v + d_{ff}) d_{emb} n_{ctx}\right) \tag{2.18}$$

To get the total cost for N layers, it just needs to be multiplied by N.

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

Knowing that, in general, $d_k \approx d_v \approx d_{emb} \approx d_{ff}$, the complexity can be simplified.

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

Two major terms emerge from the above complexity.

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

It can be noticed that the dimension of the vocabulary n_{voc} does not appear in the complexity. This dimension is indeed hidden in the embedding process.

[4] For instance, an FNN layer could be used to remember that Max Verstappen is an F1 driver although it is not written explicitly in the input context.

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 ε:

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

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 δ_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 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 σ 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, σ needs to be scaled properly, otherwise the model will not learn well. If σ 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 σ is too large, then the noise dominates the signal, and the model may struggle to learn meaningful patterns.

The loss function used is the Fisher divergence

$$\mathbb{E}\left[\left\|\varepsilon_\theta(Y) + \sigma\nabla_y \log f_Y(y)\right\|^2\right] = \int_y f_Y(y)\left\|\varepsilon_\theta(y) + \sigma\nabla_y \log f_Y(y)\right\|^2 \mathrm{d}y$$

3.3.1 Variance dependent score

To address the issue of choosing the right noise level σ, we can introduce a variance-dependent score function. Instead of using a fixed σ, 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 σ 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 what 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.

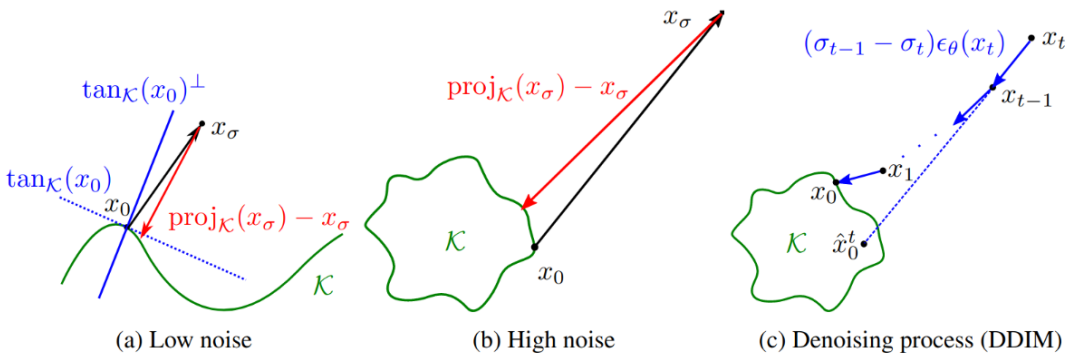


Figure 3.1: DDIM sampling example

DDIM move from a variance σ_t to a variance σ_{t-1}, like in figure 3.1.

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

3.5 Denoising with randomness

Now we want to add some randomness to the denoising process. 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 what we call the DDPM sampler (Denoising Diffusion Probabilistic Models). We have by assumption $\sigma_{t-1} < \sigma_t$. Then, because σ_{t-1} is the geometric mean of σ_t and $\sigma_{t'}$, it must be in between $\sigma_{t'} < \sigma_{t-1} < \sigma_t$. To choose η, 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 minimize the loss:

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

$E(X)$ 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:

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

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 Variational 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.

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

3.6.2 Evidence Lower Bound (ELBO)

The whole goal of diffusion models is to learn a distribution $p_\theta(x)$.

Ideally, one would like to maximize the log-likelihood

$$\max_\theta \mathbb{E}_{p_{data}(x)}[\log p_\theta(x)] \tag{3.12}$$

However, as explained above, diffusion models are formulated as *latent-variable models*, where the latent variables correspond to progressively noised versions of the data.

Using a latent trajectory $Z = x_{1:T}$, representing noisy versions of x_0 where the latent variables $z \sim Z = (x_1, \ldots, x_T)$ correspond to progressively corrupted versions of the data, one can write

$$p_\theta(x) = \int p_\theta(x, z)\, \mathrm{d}z \tag{3.13}$$

However, this integral is usually intractable.

The goal of the *ELBO* on $\log p_\theta(x)$ is to provide a tractable expression of a lower bound that can be used to optimize the model by maximizing it.

Introducing the distribution $q(z \mid x)$[1], eq.3.13 can be rewritten

$$\begin{aligned} \log p_\theta(x) &= \log \int p_\theta(x, z)\, \mathrm{d}z \\ &= \log \int q(z \mid x) \frac{p_\theta(x, z)}{q(z \mid x)}\, \mathrm{d}z \end{aligned} \tag{3.14}$$

Using the definition of expectation

$$\mathbb{E}_{p(z)}[f(z)] = \int p(z) f(z)\, \mathrm{d}z \tag{3.15}$$

expression 3.14 becomes

$$\log p_\theta(x) = \log \mathbb{E}_{q(z|x)}\left[\frac{p_\theta(x, z)}{q(z \mid x)}\right] \tag{3.16}$$

and since the log function is concave, it respects $\log \mathbb{E}[f(x)] \geq \mathbb{E}[\log f(x)]$, and the *Evidence Lower Bound* (ELBO) on $p_\theta(x)$ may be derived

$$\log p_\theta(x) \geq E_{q(z|x)}\left[\log \frac{p_\theta(x, z)}{q(z \mid x)}\right] = E_{q(z|x)}[\log p_\theta(x, z) - \log q(z \mid x)] \triangleq \mathscr{L}_{\text{ELBO}}(x) \tag{3.17}$$

In diffusion model convention, the ELBO is usually decomposed over time steps:

$$\mathscr{L}_{\text{ELBO}}(x) = \mathbb{E}_q\left[\log p_\theta(x_0 \mid x_1) + \sum_{t=2}^{T} \log p_\theta(x_{t-1} \mid x_t)\right] + \log p(x_T) \tag{3.18}$$

[1] $q(z \mid x)$ represents the *variational posterior*. It corresponds to an approximation of the posterior distribution with useful properties such as easier sampling, parametrization (often using a neural net), and the most important one, a tractable density. It is called *variational* because the definition of *ELBO* is often generalized and in some cases, $q(z \mid x)$'s parameters may be learned, hence $q(z \mid x)$ can variate. In diffusion models, however, the learning problem consists solely in learning the reverse process $p_\theta(z \mid x)$, while $q(z \mid x)$ is fixed

3.6.3 KL Divergence

The KL divergence is a meaningful tool to understand how maximizing the *ELBO* leads to learning the right distribution.

It is defined as

$$D_{KL}(p\|q) = \mathbb{E}_{X\sim p}\left[\log\frac{p(x)}{q(x)}\right] = \mathbb{E}_{X\sim p}[\log p(x) - \log q(x)] \tag{3.19}$$

which somehow looks like the *ELBO* derived in the previous section. It can be thought of as a measure of the discrepancy between any two distributions.

Recalling that the joint and posterior distributions are related

$$p_\theta(z \mid x) = \frac{p_\theta(x,z)}{p_\theta(x)} \tag{3.20}$$

and using the definition 3.19. The following *KL Divergence* stands

$$\begin{aligned} D_{KL}(q(z \mid x)\|p_\theta(z \mid x)) &= \mathbb{E}_{q(z|x)}\left[\log\frac{q(z \mid x)}{p_\theta(z \mid x)}\right] \\ &= \mathbb{E}_{q(z|x)}\left[\log\frac{q(z \mid x)p_\theta(x)}{p_\theta(x,z)}\right] \\ &= \mathbb{E}_{q(z|x)}[\log q(z \mid x) + \log p_\theta(x) - \log p_\theta(x,z)] \\ &= \underbrace{\mathbb{E}_{q(z|x)}[\log q(z \mid x) - \log p_\theta(x,z)]}_{-\mathscr{L}_{\text{ELBO}}(x)} + \log p_\theta(x) \end{aligned} \tag{3.21}$$

where the last equality stands because $\log p_\theta(x)$ is constant in z.

Hence, the final expression is

$$D_{KL}(q(z \mid x)\|p_\theta(z \mid x)) = \log p_\theta(x) - \mathscr{L}_{\text{ELBO}}(x) \tag{3.22}$$

Although $p_\theta(x)$ is not a tractable expression, and neither is the *KL Divergence*, it provides an insight into how maximizing the lower bound leads to approximating the underlying distribution $p_\theta(x)$ efficiently.

As $\mathscr{L}_{p_\theta(x)}(x)$ is a lower bound on $p_\theta(x)$, D_{KL} is always non-negative, which is a required condition for a measure. D_{KL} measures the error introduced by approximating the true posterior distribution during inference.

Eventually, eq.3.22 explains where the name *variational inference* comes from. The *variational* aspect is due to the optimized nature of $q(z \mid x)$'s parameters, while the *inference* keyword is because this expression transforms the learning process into an approximation of the **Bayesian inference**. The goal is to **infer** a variational posterior distribution $q(z \mid x)$ that minimizes the distance to the true posterior $p_\theta(z \mid x)$.

This identity can be seen as a *certificate of optimality*.

3.6.4 Gaussian ELBO

We can compute the ELBO in the case of Gaussian distributions, like with the variational 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:

$$2D_{KL}((Y|X=x)\|Z) = \|E_\mu(X)\|_2^2 + \|E_\sigma(X)\|_2^2 - \left(\sum_{i=1}^{r}\log((E_\sigma(X))_i^2) + r\right) \tag{3.23}$$

And for the first term of the ELBO:

$$\begin{aligned}
&\mathbb{E}\left[\log(f_{X|Z}(x|Y))\right] \\
&= \mathbb{E}\left[\log(f_{X|Z}(x|E_\mu(X)+\varepsilon_2\odot E_\sigma(X)))\right] \\
&= -\frac{\log(2\pi)}{2} + \mathbb{E}\left[\|\mathrm{Diag}(D_\sigma(E_\mu(X)+\varepsilon_2\odot E_\sigma(X)))^{-1}\left(x - D_\mu(D_\sigma(E_\mu(X)+\varepsilon_2\odot E_\sigma(X)))\right)\|_2^2\right]
\end{aligned} \tag{3.24}$$

3.6.5 Monte-Carlo sampling

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

$$\mathbb{E}\left[\log(f_{X|Z}(x|Y))\right] \approx \frac{1}{L}\sum_{i=1}^{L}\log(f_{X|Z}(x|E_\mu(X)+\epsilon_i\odot E_\sigma(X))) \tag{3.25}$$

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

$$\mathbb{E}\left[\log(f_{X|Z}(x|Y))\right] \approx -\frac{\log(2\pi)}{2} + \frac{1}{L}\sum_{i=1}^{L}\|x - D_\mu(E_\mu(x)+\epsilon_i)\|_2^2 \tag{3.26}$$

3.6.6 Maximum Likelihood Estimator with variational Auto-Encoder

Remember that 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:

$$\sum_x \log(f_X(x)) \geq \sum_x -D_{KL}((Y|X=x)\|Z) + \mathbb{E}[\log(f_{X|Z}(x|Y))] \tag{3.27}$$

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

3.6.7 Denoising Auto-Encoder

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

- Auto-Encoder $D(E(X))$;
- Variational 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 $X=D(E(Z))$:

$$-\log(f_X(x)) \leq D_{KL}((Y|X=x)\|Z) - \mathbb{E}[\log(f_{X|Z}(x|Y))] \tag{3.28}$$

3.7 Conditioned Diffusion Models

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

3.8 Classifier-Free Guidance

To improve the conditioned diffusion for multi-modal distributions, we can use a classifier but it also needs 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 conditioned (with condition τ) and unconditioned diffusion model and combine them with:

$$\bar{\epsilon}_t = \lambda \epsilon_\theta(x_t, \sigma_t, \tau) + (1-\lambda)\epsilon_\theta(x_t, \sigma_t) \qquad \lambda > 1 \tag{3.29}$$

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:

$$\bar{\epsilon}_t = \frac{1}{N}\sum_{i=1}^{N} v_i^{-1}(\epsilon_\theta(v_i(x_t), \sigma_t, \tau)) \tag{3.30}$$

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

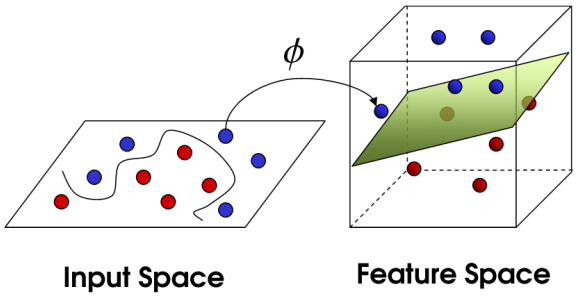


Figure 4.1: Illustration of kernels

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 $M > 0 \in \mathbb{R}^{n \times n}$ with the scalar product $\langle x, y \rangle_M = x^T M y$. 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 ϕ 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 ϕ: 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 (ϕ 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,

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

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 16M 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 $N \ll h$.

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 continuous 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

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

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

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

Then, the dimension of the out space $H = \mathbb{R}^\ell$ is $\ell = \binom{n+d-1}{d}$. Of course, with the kernel trick, the embedding ϕ is never actually computed in practice, as the only thing that we need is the scalar product.
From the kernel, we can recover automatically the feature map $\psi(x) = k(x, \cdot)$. For example, with $d = 2$,

$$\forall x = \begin{pmatrix} x_1 & x_2 \end{pmatrix}^T, \psi(x_1, x_2) : \mathbb{R}^2 \to \mathbb{R} : (y_1, y_2) \to x_1^2 y_1^2 + x_2^2 y_2^2 + 2 x_1 x_2 y_1 y_2 \tag{4.17}$$

and we have some examples:

$$\begin{aligned} &\psi(1,0) : \mathbb{R}^2 \to \mathbb{R} : (y_1, y_2) \to y_1^2 \\ &\psi(1,-1) : \mathbb{R}^2 \to \mathbb{R} : (y_1, y_2) \to y_1^2 + y_2^2 - 2 y_1 y_2 \end{aligned} \tag{4.18}$$

We can see that $\psi(1,-1) = 2\psi(1,0) + 2\psi(0,1) - \psi(1,1)$, and we can actually show that $\psi(x_1, x_2)$ can always be expressed as a linear combination of $\{\psi(1,0), \psi(0,1), \psi(1,1)\}$, meaning that there is an equivalence between this feature map and the kernel function $\phi(x)$ defined in the initial example.

4.5 Operations on kernel

- The sum of two kernels is a kernel: if $k_1 : X \times X \to \mathbb{R}$ and $k_2 : X \times X \to \mathbb{R}$ are two kernels, then so is $k = c_1 k_1 + c_2 k_2$, $c_1, c_2 \geq 0$. This is because if $K_1 \succeq 0$ and $K_2 \succeq 0$, then $K \succeq 0$.

- The product of two kernels is a kernel: if $k_1 : X \times X \to \mathbb{R}$ and $k_2 : X \times X \to \mathbb{R}$ are two kernels, then so is $k = k_1 k_2$. This is because if $K_1 \succeq 0$ and $K_2 \succeq 0$, then $K = K_1 \odot K_2 \succeq 0$.

- The positive-coefficient polynomial function of a kernel is a kernel: if $k : X \times X \to \mathbb{R}$ is a kernel, then so is $p(k)$. This is because of the two rules above.

- For any scalar map $c : X \to \mathbb{R}$, if $k(x, y)$ is a kernel, then so is $c(x)k(x, y)c(y)$. This is proven by $K \succeq 0 \Longrightarrow CKC \succeq 0$ for $C = diag(c(x_i))$.

4.6 Gaussian kernel

If $X \subseteq \mathbb{R}^n$, then the Gaussian kernel is

$$(x, y) \to k(x, y) = e^{-\frac{\|x-y\|^2}{2\sigma^2}} \tag{4.19}$$

This kernel has infinite dimension: we can reexpress it using a Taylor development:

$$k(x, y) = e^{-\frac{\|x-y\|^2}{2\sigma^2}} = e^{-\frac{\|x\|^2}{2\sigma^2}} \sum_{d=0}^{\infty} \frac{1}{d!} \frac{(x^T y)^d}{\sigma^{2d}} e^{-\frac{\|y\|^2}{2\sigma^2}} \tag{4.20}$$

As it is an infinite sum of polynomial kernels of degree d, it has indeed infinite dimension. Obviously, $k(x, x) = 1$, and that means that all points $\phi(x)$ are on the unit sphere in the space H, and $k(x, y) = \cos(\theta)$, θ being the angle between $\phi(x)$ and $\phi(y)$ on the sphere.

If $\|x - y\| \ll \sigma$, then we can do some approximations with Taylor developments:

$$\begin{aligned} \cos(\theta) &\approx 1 - \frac{\theta^2}{2} \\ e^x \approx 1 + x \implies e^{-\frac{\|x-y\|^2}{2\sigma^2}} &\approx 1 - \frac{\|x-y\|^2}{2\sigma^2} \end{aligned} \tag{4.21}$$

Then, for close points x, y, there is a relation connecting the norm in both spaces:

$$\theta = \|\phi(x) - \phi(y)\|_H \approx \frac{\|x - y\|_{\mathbb{R}^n}}{\sigma} \tag{4.22}$$

This means that the feature map preserves the local geometry of X, up to a scaling factor. Furthermore, if $\|x - y\|_{\mathbb{R}^n} \gg \sigma$, then $\langle \phi(x), \phi(y) \rangle_H \approx 0$, meaning that the feature map acts like a one-hot encoding on distant points. Points far away from each other in $\mathbb{R}^n$ are orthogonal in H.

4.7 Random Fourier Features

The main limitation of kernels is the computation of the kernel matrix K: it requires $\mathcal{O}(N^2)$ in storage for N data points, and $\mathcal{O}(N^2)$ (or even $\mathcal{O}(N^3)$) in computation time.
The kernel trick is a way to implicitly work in a high dimensional feature space. Here, we want to explicitly approximate the feature space to reduce the computational complexity of kernels. For that, we construct a mapping $z : \mathbb{R}^n \to \mathbb{R}^D$, with D reasonably small, such that

$$\forall x, y \in X, \qquad k(x, y) = \langle \phi(x), \phi(y) \rangle \approx \langle z(x), z(y) \rangle \tag{4.23}$$

The kernel matrix of z is $K'_{ij} = k'(x, y) = \langle z(x), z(y) \rangle$. Here, instead of computing this kernel matrix, we compute the embedding, i.e. the opposite of the kernel trick:

- Kernel trick: embed the points into high or infinite dimension space implicitly: $\mathcal{O}(N^2)$;
- Kernel approximation: embed the points into a small (finite) space explicitly: $\mathcal{O}(Nd)$.

4.7.1 Recognizing a kernel

Given a function $k : X \times X \to \mathbb{R} : (x, y) \to k(x, y)$, we can check that it is a kernel using one of the following methods:

- Check for every set $\{x_i\}_{i=1}^N$ that K is symmetric definite-positive;
- Find an embedding $\phi : X \to H$ such that $k(x, x') = \langle \phi(x), \phi(x') \rangle$ for all $x, x' \in X$;
- Recognize that k is a product, combination, function of known kernels;
- Use a specific theorem, e.g. Bochner (see later).

4.7.2 Shift-invariant kernels and Bochner's theorem

A kernel k is shift-invariant if $k(x, y) = k(0, x - y)$.

Theorem 4.2. A shift-invariant function $k(x, y)$ is a kernel iff $f(u) = c\mathbb{E}_{\omega \sim \mathscr{P}}[e^{i\omega^T u}]$, where $f(u) = f(x - y) = k(0, x - y)$, for some distribution $\mathscr{P}$ and for some constant $c \in \mathbb{R}_0^+$.

This is in fact connected to the Fourier transform of p:

$$k(x, y) = k(0, x - y) = f(x - y) = \int_\omega e^{i\omega^T (x-y)} p(\omega) d\omega = \mathbb{E}_p[e^{i\omega^T (x-y)}] \tag{4.24}$$

→ Note: the polynomial kernels are not shift invariant, but the Gaussian kernel is.

As $\mathscr{P}(\omega)$ is a probability function form which we can get a sample, we can use a Monte-Carlo method to approximate the kernel function: for a sample of D points,

$$k'(x, y) = \frac{1}{D} \sum_{j=1}^{D} e^{i\omega_j^T (x-y)} = \frac{1}{D} \sum_{j=1}^{D} e^{i\omega_j^T x} e^{-i\omega_j^T y} = z(x)^* z(y) \tag{4.25}$$

Then,

$$z : \mathbb{R}^N \to \mathbb{C}^D : x \to \frac{1}{\sqrt{D}} \begin{pmatrix} e^{i\omega_1^T x} \\ \vdots \\ e^{i\omega_D^T x} \end{pmatrix} \tag{4.26}$$

If the probability distribution $\mathscr{P}(\omega)$ is symmetric, $p(\omega) = p(-\omega)$, we can extend the dimension of z, and stay in the real space:

$$\begin{aligned} k(x, y) = \mathbb{E}_{\omega \sim \mathscr{P}}[e^{i\omega^T (x-y)}] &= \int_\omega [cos(\omega^T (x - y)) + i \sin(\omega^T (x - y))] p(\omega) d\omega \\ &= \int_\omega \cos(\omega^T (x - y)) p(\omega) d\omega + i \underbrace{\int_\omega \sin(\omega^T (x - y)) p(\omega) d\omega}_{=0 \text{ because odd function}} \\ &= \mathbb{E}_{\omega \sim \mathscr{P}}[\cos(\omega^T (x - y))] = \mathbb{E}_{\omega \sim \mathscr{P}} \left[\begin{pmatrix} \cos(\omega^T x) \\ \sin(\omega^T x) \end{pmatrix}^T \begin{pmatrix} \cos(\omega^T y) \\ \sin(\omega^T y) \end{pmatrix} \right] \\ &\implies k'(x, y) = \frac{1}{D} \sum_{j=1}^{D} \begin{pmatrix} \cos(\omega_j^T x) \\ \sin(\omega_j^T x) \end{pmatrix}^T \begin{pmatrix} \cos(\omega_j^T y) \\ \sin(\omega_j^T y) \end{pmatrix} \\ &\implies z : X \to \mathbb{R}^{2D} : x \to z(x) = \frac{1}{\sqrt{D}} \begin{pmatrix} \cos(\omega_1^T x) \\ \sin(\omega_1^T x) \\ \vdots \\ \cos(\omega_D^T x) \\ \sin(\omega_D^T x) \end{pmatrix} \end{aligned} \tag{4.27}$$

We can also add some uniform shift on $[0, 2\pi]$, to get rid of the sine terms:

$$\begin{aligned} k(x,y) &= \mathbb{E}_{\omega\sim\mathscr{P}}\left[\cos(\omega^\top(x-y))\right] \\ &= \mathbb{E}_{\substack{\omega\sim\mathscr{P} \\ b\sim\text{Uni}[0,2\pi]}}\left[\cos(\omega^\top(x-y))\right] + \underbrace{\mathbb{E}_{\substack{\omega\sim\mathscr{P} \\ b\sim\text{Uni}[0,2\pi]}}\left[\cos(\omega^\top(x+y)+2b)\right]}_{=0 \text{ by computing the integral}} \\ &= \mathbb{E}_{\substack{\omega\sim\mathscr{P} \\ b\sim\text{Uni}[0,2\pi]}}\left[2\cos(\omega^T x+b)\cos(\omega^T y+b)\right] \end{aligned} \tag{4.28}$$

Let us pick D points $\omega_j \sim \mathscr{P}$ and $b_j \sim \text{Uni}[0, 2\pi]$. We get the kernel:

$$\begin{aligned} k'(x,y) &= \frac{1}{D}\sum_{j=1}^{D} 2\cos(\omega_j^T x+b_i)\cos(\omega_j^T y+b_j) = \langle z(x), z(y)\rangle \\ &\implies z: X\to\mathbb{R}^D : x\to z(x) = \sqrt{\frac{2}{D}}\begin{pmatrix}\cos(\omega_1 x+b_1)\\ \vdots \\ \cos(\omega_D x+b_D)\end{pmatrix} \end{aligned} \tag{4.29}$$

The form of $z(x)$ is what we call the Random Fourier Features.
It is also possible to find the distribution $\mathscr{P}$ from the kernel by doing the inverse Fourier transform:

$$k(x,y) = \mathscr{F}_\omega(p(\omega)) \implies \mathscr{P}(\omega) = \mathscr{F}^{-1}(k) \tag{4.30}$$

For example, for the Gaussian kernel, $k(x,y) = f(u) = e^{-\frac{\|u\|^2}{2\sigma^2}}$ and

$$\begin{aligned} p(\omega) &= \frac{1}{(2\pi)^n}\int_u e^{-\frac{\|u\|^2}{2\sigma^2}} e^{-i\omega^T u}\,du = \frac{1}{(2\pi)^n} e^{-\frac{\sigma^2}{2}\|\omega\|^2}\int_u e^{-\frac{1}{2\sigma^2}\|u+i\sigma^2\omega\|^2}\,du \\ &= \frac{1}{(2\pi)^n} e^{-\frac{\sigma^2}{2}\|\omega\|^2}(2\pi)^{n/2}\sigma^n = \frac{1}{(2\pi)^n}\sigma^n e^{-\frac{\sigma^2}{2}\|\omega\|^2} \\ &\implies p\sim\mathscr{N}(0,\frac{1}{\sigma^2}I_n) \end{aligned} \tag{4.31}$$

and we can estimate the error of the kernel approximation:

$$P\left[|k'(x,y)-k(x,y)|\geq\epsilon\right] = P\left[|z^T(x)z(y)-k(x,y)|\geq\epsilon\right] \leq 2e^{-D\epsilon^2/4} \tag{4.32}$$

We say that the kernel k' is probably approximately correct (PAC).

4.8 Applications of kernels

4.8.1 Kernel Ridge Regression

The classical Ridge regression is a linear regression with a penalty term on the slope of the line:

$$\alpha^* = \arg\min_{\alpha\in\mathbb{R}^N}\frac{1}{2}\|X\alpha-y\|^2+\lambda\frac{1}{2}\alpha^T\alpha \tag{4.33}$$

where X is the data matrix. The kernel Ridge regression uses the kernel matrix K instead:

$$\alpha^* = \arg\min_{\alpha\in\mathbb{R}^N}\frac{1}{2}\|K\alpha-y\|_2^2+\lambda\frac{1}{2}\alpha^T K\alpha \tag{4.34}$$

where $K_{ij} = k(x_i, x_j)$. We can generalize further the loss function and the regularization function:

$$\alpha^* = \arg\min_{\alpha \in \mathbb{R}^N} \mathscr{L}(x_i, y_i, f(x_i)) + \lambda\Omega(\|f\|_H^2) \tag{4.35}$$

where $\mathscr{L}$ is the loss function and Ω the regularization function. This allows to find nonlinear regressions more easily.

Theorem 4.3. Representer theorem. Let H be a RKHS associated with a kernel $k(x, y)$, with feature map $\phi(x)$. The solution of the problem

$$\arg\min_{f \in H} \mathscr{L}(x_i, y_i, f(x_i)) + \lambda\Omega(\|f\|_H^2) \tag{4.36}$$

is a linear combination of the featured data:

$$f^*(\cdot) = \sum_{i=1}^{N} \alpha_i k(\cdot, x_i) = \sum_{i=1}^{N} \alpha_i \phi(x_i) \tag{4.37}$$

and

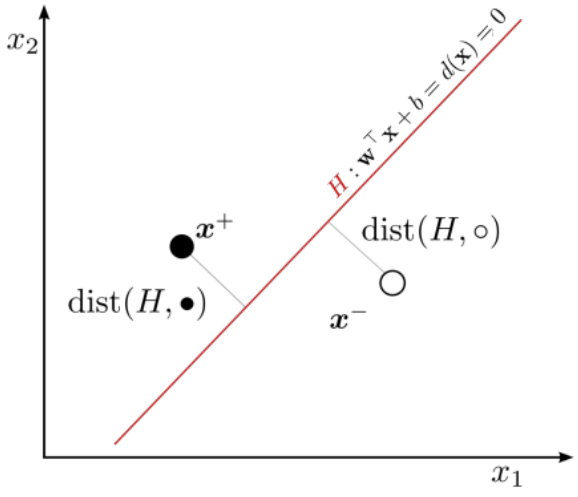
$$\begin{aligned} f(x_i) &= \sum_{j=1}^{N} \alpha_j k(x_i, x_j) = \sum_{j=1}^{N} \alpha_j K_{ij} = (K\alpha)_i = K_i^T \alpha \\ \|f\|_H^2 &= \left\| \sum_{i=1}^{N} \alpha_i \phi(x_i) \right\|_H^2 = \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j \langle \phi(x_i), \phi(x_j) \rangle_H = \alpha^T K \alpha \end{aligned} \tag{4.38}$$

This means that the minimisation on the function f is equivalent to the minimisation on the vector α (much easier).

4.8.2 Support Vector Machines

A classical linear classifier searches for parameters w and b to define a hyperplane that cuts the space between two classes: $d(x) = w^T x + b$. If $d(x) > 0$, $y_{pred} = 1$ and if $d(x) < 0$, $y_{pred} = -1$. There is thus the constraint that $y_i(w^T x_i + b) > 0$. There is usually a relatively large space in which the plane can go. To determine the "best" hyperplane that classifies correctly, we can decide to take the one that maximizes the margin to the closest point of each class:

$$\text{margin} = \min_{i, y_i = 1} \frac{|w^T x_i^{(1)} + b|}{\|w\|_2} + \min_{i, y_i = -1} \frac{|w^T x_i^{(-1)} + b|}{\|w\|_2} \tag{4.39}$$



Then, the optimal parameters are

$$\arg\max_{w,b} \frac{\min_i w^T x_i^{(-1)} + b}{\|w\|_2} - \frac{\max_i w^T x_i^{(1)} + b}{\|w\|_2} \qquad \text{s.t. } y_i(w^T x_i + b) > 0 \,\forall i \in \{1,\dots,N\} \tag{4.40}$$

This formulation is invariant to the scaling of the parameters: reformulating the constraints as

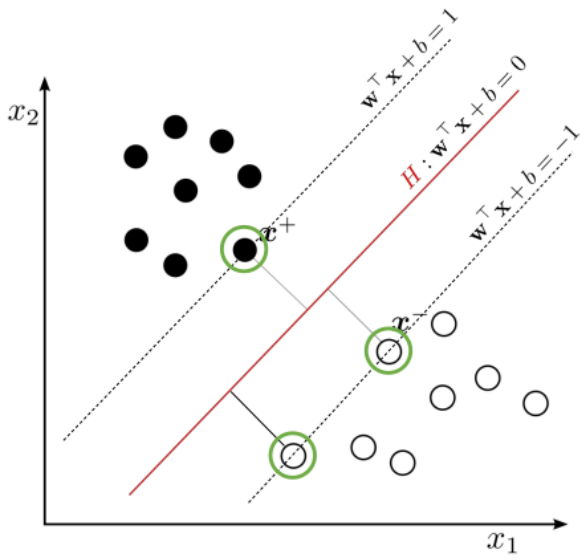
$$y_i(w^T x_i + b) \geq \epsilon \,\forall i, \epsilon > 0 \iff y_i\left(\frac{w}{\epsilon}^T x_i + \frac{b}{\epsilon}\right) \geq 1 \,\forall i, \epsilon > 0 \tag{4.41}$$

The support vectors are the points for which the constraints are tight:

$$\begin{cases} w^T x_i^{(1)} = 1 \\ w^T x_i^{(-1)} = -1 \end{cases} \tag{4.42}$$

In that case, the margin is $2/\|w\|_2$ and the problem to find the parameters of the support vectors are the solution of

$$\arg\max_{w,b} \frac{2}{\|w\|_2} \qquad \text{s.t. } y_i(w^T x_i + b) \geq 1 \qquad \forall i \in \{1,\dots,N\} \tag{4.43}$$



We can simplify the objective to $\arg\min_{w,b} \frac{1}{2} w^T w$. In that case, the problem becomes convex (quadratic) and only involves the basic scalar product. We call this the hard-margin SVM.

Moreover, the margin should be completely determined by the data, meaning that we can express w and b as a linear combination of the data:

$$w^* = \sum_{i=1}^{N} \gamma_i^* x_i \tag{4.44}$$

As the margin constraints are tight on the support vectors only, these are the only weights that are nonzero: $\gamma_i^* \neq 0$ for x_i a support vector. Then,

$$\begin{aligned} w^* &= \sum_{\substack{i=1 \\ \gamma_i^* \neq 0}}^{N} \gamma_i^* x_i \\ b^* &= y_i - w^{*T} x_i \end{aligned} \tag{4.45}$$

Kernel SVM

We can generalize the hard-margin SVM to kernels by changing the scalar product:

$$\begin{aligned} &\arg\min_{f_w \in H} \frac{1}{2}\langle f_w, f_w\rangle_H \\ &\qquad y_i(\langle f_w, f_{x_i}\rangle_H + b) \geq 1 \qquad \forall i \in \{1,\dots,N\} \end{aligned} \tag{4.46}$$

where $f_w(\cdot) = \phi(w) = k(w,\cdot)$ and $H = \{f_w : \mathbb{R}^n \to \mathbb{R} : f_w(\cdot) = k(w,\cdot) \text{ s.t. } w \in \mathbb{R}^n\}$. The scalar product is

$$\langle f_w, f_{x_i}\rangle_H = \langle \phi(w), \phi(x_i)\rangle_H = k(w, x_i) = f_{x_i}(w) \tag{4.47}$$

As we are minimizing on a function, the search space is infinite. However, as previously, we can express the solution as a linear combination of the embeddings of the data points and work in a finite-dimensional space. We consider the bias to be fixed for now.

$$\begin{aligned} &\arg\min_{\gamma \in \mathbb{R}^N} \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \gamma_i \gamma_j \langle f_{x_i}, f_{x_j}\rangle_H \\ &\qquad y_i\left(\sum_{j=1}^{N} \gamma_j \langle f_{x_j}, f_{x_i}\rangle_H + b\right) \geq 1 \qquad \forall i \in \{1,\dots,N\} \end{aligned} \tag{4.48}$$

The kernel matrix appears and we can compute it beforehand to simplify the optimisation:

$$\arg\min_{\gamma \in \mathbb{R}^N} \frac{1}{2}\gamma^T K \gamma \qquad \text{s.t.} \qquad y_i(K_i^T \gamma + b) \geq 1 \qquad \forall i \in \{1,\dots,N\} \tag{4.49}$$

This problem is convex since K is positive-definite. The number of variables is small compared to the infinite-dimensional space of functions from the initial situation.
Once the γ^* are found, we can easily compute the optimal bias:

$$b^* = y_i - \sum_{\substack{j=1 \\ \gamma_j^* \neq 0}}^{N} \gamma_j^* \phi(x_j)^T \phi(x_i) = y_i - \sum_{\substack{j=1 \\ \gamma_j^* \neq 0}}^{N} \gamma_j^* k(x_j, x_i) \tag{4.50}$$

4.8.3 Kernel PCA

The classical PCA consists in selecting the k features with the most variance in a space of dimension n. We can show (not done here) that the k principal components of the dataset are the k eigenvectors of the covariance matrix that correspond to the k largest eigenvalues of that matrix. The algorithm is the following:

Algorithm 1 PCA

1: Dataset: $X = \begin{pmatrix} x_1 & x_2 & \dots & x_N \end{pmatrix} \in \mathbb{R}^{n \times N}$
2: Centered data: $\tilde{X} = X - X\frac{1_{N\times N}}{N}$
3: Empirical Covariance matrix: $C = \frac{1}{N}\tilde{X}\tilde{X}^T \in \mathbb{R}^{n\times n}$
4: Compute the k principal components v:

$$C v_k = \lambda_k v_k \qquad \text{such that} \qquad v_k^T v_k = 1 \tag{4.51}$$

5: Project the data on the k Principal components:

$$y_k = V_k^T y \tag{4.52}$$

The dual of this problem (this is optimisation, although the form is simplified here) is

$$X^T X\alpha = \lambda\alpha \tag{4.53}$$

and we find back the principal components by doing the product $v = X\alpha$.

The kernel PCA is almost the same, but it applies a kernel $\phi : \mathbb{R}^n \to \mathbb{R}^h$ on the data before applying the algorithm:

Algorithm 2 Kernel PCA

1: Lifted dataset: $\Phi = \begin{pmatrix}\phi(x_1) & \dots & \phi(x_N)\end{pmatrix} \in \mathbb{R}^{h\times N}$
2: The kernel matrix is $K = \Phi^T\Phi \in \mathbb{R}^{N\times N}$
3: Center the lifted data: $\tilde{\Phi} = \Phi - \Phi\frac{\mathbb{1}_{N\times N}}{N}$;
4: The centered kernel matrix is $\tilde{K} = \tilde{\Phi}^T\tilde{\Phi}$
5: Compute the empirical Covariance matrix: $C = \frac{1}{N}\tilde{\Phi}\tilde{\Phi}^T \in \mathbb{R}^{h\times h}$
6: Compute the k principal components v:

$$\tilde{K}\alpha_k = \lambda_k\alpha_k \qquad v_k = \frac{1}{\sqrt{\lambda_k}}\alpha_k \tag{4.54}$$

7: Project the data on the k Principal components:

$$y_k = V_k^T y \tag{4.55}$$

kPCA coordinates

Let $f = \phi(x) \in \mathscr{H}$ be any function embedding of some point $x \in \mathbb{R}^n$. We can express it as a linear combination of the $K \leq N$ principal components $v_k \in \mathscr{H}$:

$$f = \phi(x) = \sum_{k=1}^{K} \langle v_k, f\rangle_{\mathscr{H}} v_k \tag{4.56}$$

And we can express the principal components in terms of the embedding and the k-th eigenvector of the kernel matrix:

$$v_k = \sum_{i=1}^{N} (\alpha_k)_i \phi(x_i) \tag{4.57}$$

From this, the inner product is f is the projection of f on v_k:

$$\langle v_k, f\rangle_{\mathscr{H}} = \sum_{i=1}^{N} (\alpha_k)_i \langle\phi(x_i), \phi(x)\rangle_{\mathscr{H}} = \sum_{i=1}^{N} (\alpha_k)_i k(x, x_i) \tag{4.58}$$

In the next algorithm, we use the index i on the principal components to have clear notations.
If we have the kernel function $k(\cdot,\cdot)$, the algorithm 2 becomes

Algorithm 3 Kernel PCA with kernel function

1: Choose a kernel $k(x, y)$ and compute the kernel matrix K:

$$k(x, y) = \langle \phi(x), \phi(y) \rangle_{\mathscr{H}} \qquad K_{ij} = k(x_i, x_j) \tag{4.59}$$

2: Compute the centered kernel $\tilde{k}(x, y)$ and the centered kernel matrix $\tilde{K}$:

$$\begin{aligned} \tilde{k}(x, y) &= k(x, y) - \frac{1}{N} \sum_{i=1}^{N} (k(x_i, x) + k(x_i, y)) + \frac{1}{N^2} \mathbb{1}_N^T K \mathbb{1}_N \\ \tilde{K} &= K - \frac{\mathbb{1}_{N \times N}}{N} K - K \frac{\mathbb{1}_{N \times N}}{N} + \frac{\mathbb{1}_{N \times N}}{N} K \frac{\mathbb{1}_{N \times N}}{N} \end{aligned} \tag{4.60}$$

3: Compute k scaled eigenvectors α_i and eigenvalues $\tilde{\lambda}_i$ of the kernel matrix:

$$\tilde{K} \alpha_i = \tilde{\lambda}_i \alpha_i \qquad \alpha_i^T \alpha_i = \frac{1}{\tilde{\lambda}_i} \tag{4.61}$$

4: Project the new data on the i Principal components:

$$y_k = \tilde{\phi}(x)_k = \sum_{j=1}^{N} \alpha_{1:i,j} \tilde{k}(x_j, x) \tag{4.62}$$

kPCA is better than the classical PCA because it does not simply project the dataset; it completely transforms it and makes it linearly separable.

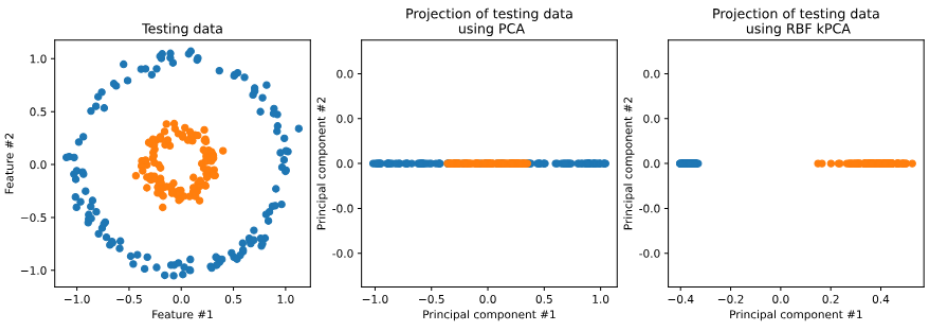


Figure 4.2: PCA vs kPCA