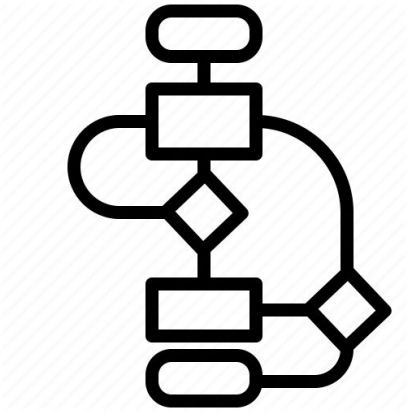


LINMA2472 - Algorithm in data science

Simon Desmidt

Academic year 2025-2026 - Q1

UCLouvain

Contents

Automatic differentiation

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

1.1 Chain rule

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

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

Let's define:

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

We thus get:

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

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

1.2 Forward differentiation

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

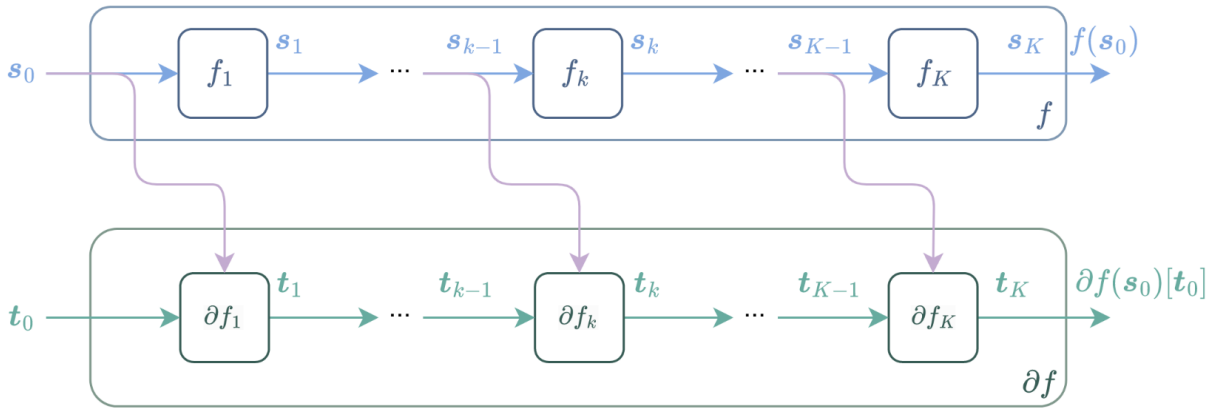


Figure 1.1: Forward differentiation

And it can be computed with the following recurrence relation:

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

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

1.3 Backward differentiation

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

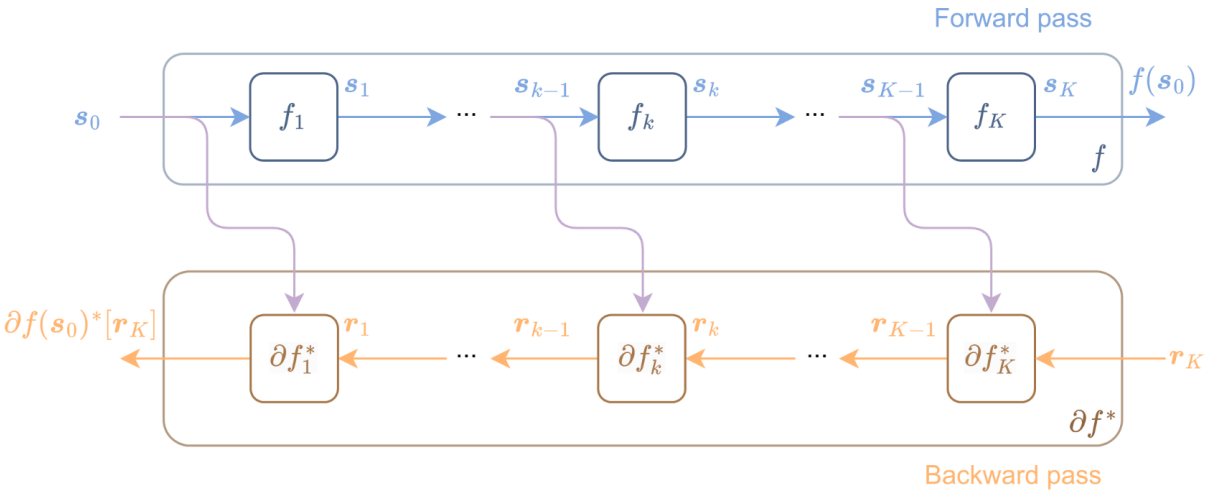


Figure 1.2: Backward differentiation

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

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

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

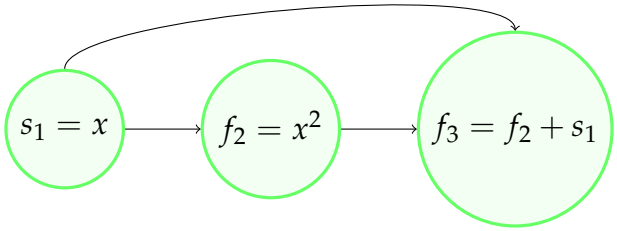
1.4 Computational graph and multivariate differentiation

1.4.1 Computational graph

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

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

The computational graph is:



1.4.2 Multivariate differentiation

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

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

following the chain rule, we have:

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

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

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

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

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

Then we compute the intermediate values in a forward pass:

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

Finally we get:

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

1.5 Jacobian computation

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

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

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

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

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

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

1.6 Memory usage

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

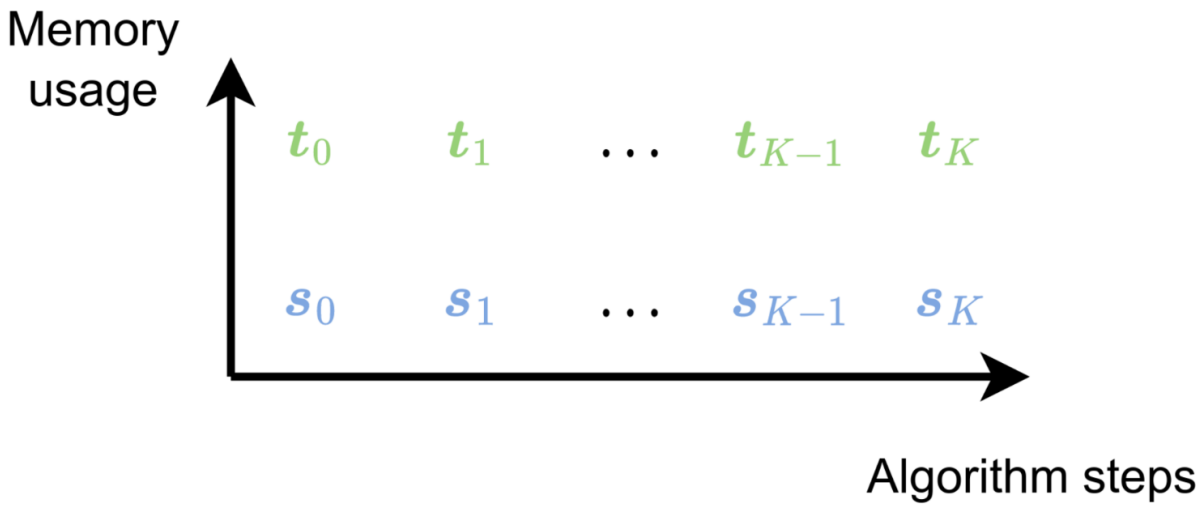


Figure 1.3: Forward mode memory usage

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

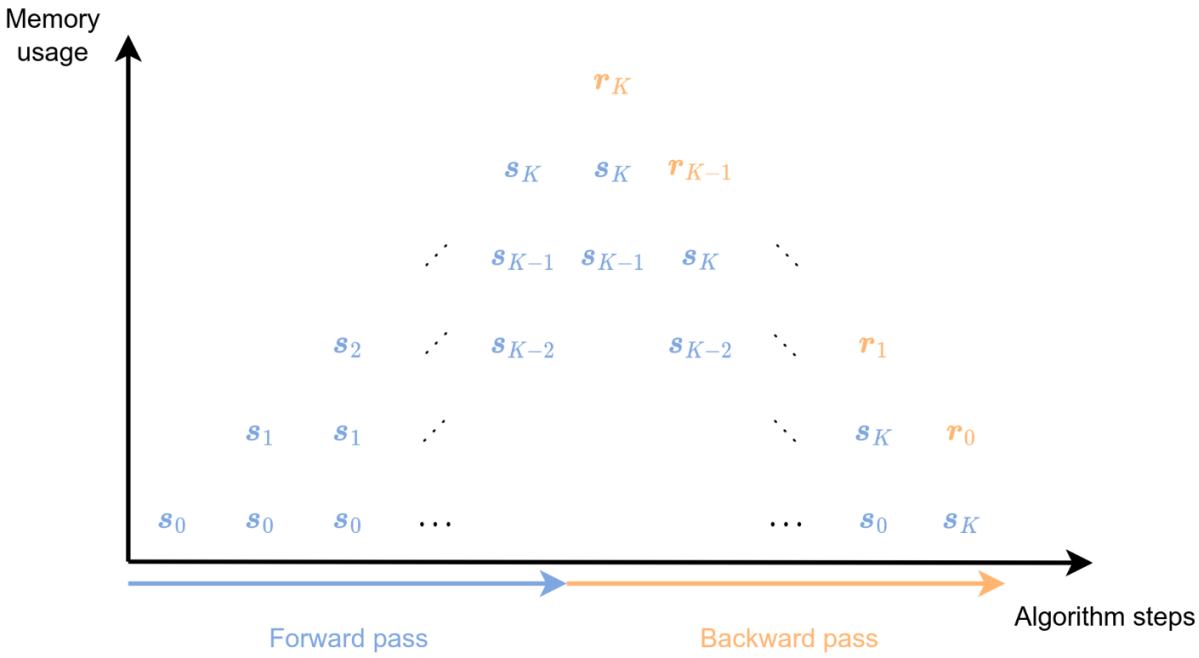


Figure 1.4: Backward mode memory usage

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

1.7 Second order AD

And what happens when we take a look at the second order derivative? Remember that we can compute the Hessian $\nabla^2 f(x)$ like this:

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

Or like this:

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

We can easily see that with this definition, we can use an AD for the gradient and for the Jacobian to compute the Hessian. And because we can separate these two computations in distinct AD, it means that we are not forced to use the same mode for both. So we can do either forward or backward for both computations without taking care of the mode used for the other computation.

Based on the chain rule, let's rewrite $\frac{\partial^2 (f_2 \circ f_1)}{\partial x_i \partial x_j}$ into something more suitable for AD (consider ∂f as the gradient of f):

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

Introducing the variables $\mathbf{J}_k = \partial f_k$ and $\mathbf{H}_{kj} = \frac{\partial}{\partial x_j}\mathbf{J}_k = \partial^2 f_k \frac{\partial f_{k-1}}{\partial x_j}$, and so we get:

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

We can now define four different ways to compute the Hessian depending on the mode used for each computation.

1.7.1 Forward on forward

Define $Dual(s_1, t_1)$ with $s_1 = Dual(f_1(x), \frac{\partial f_1}{\partial x_j})$ and $t_1 = Dual(\frac{\partial f_1}{\partial x_i}, \frac{\partial^2 f_1}{\partial x_i \partial x_j})$ then we can have this algorithm:

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

$$\begin{aligned}
t_2 &= \mathbf{J}_{f_2}(s_1)t_1 \\
&= Dual(\mathbf{J}_2, \mathbf{H}_{2j}) Dual(\frac{\partial f_1}{\partial x_i}, \frac{\partial^2 f_1}{\partial x_i \partial x_j}) \\
&= Dual(\mathbf{J}_2 \frac{\partial f_1}{\partial x_i}, \mathbf{J}_2 \frac{\partial^2 f_1}{\partial x_i \partial x_j} + \mathbf{H}_{2j}\frac{\partial f_1}{\partial x_i})
\end{aligned} \tag{1.19}$$

4. Repeat

All that can be resumed as these two equations with $g_k(x) = f_k \circ \cdots \circ f_1$:

$$\begin{cases} s_k &= Dual(g_k(x), \frac{\partial g_k}{\partial x_j}) \\ t_k &= Dual(\frac{\partial g_k}{\partial x_i}, \frac{\partial^2 g_k}{\partial x_i \partial x_j}) \end{cases} \tag{1.20}$$

1.7.2 Forward on reverse

First the forward pass works the same as forward on forward, given $s_1 = Dual(f_1(x), \frac{\partial f_1}{\partial x_j})$

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

Then the backward pass, given $r_2 = Dual((r_2)_1, (r_2)_2)$ compute:

$$\begin{aligned} r_2\mathbf{J}_2 &= Dual((r_2)_1, (r_2)_2) Dual(\mathbf{J}_2, \mathbf{H}_{2j}) \\ &= Dual((r_2)_1\mathbf{J}_2, (r_2)_1\mathbf{H}_{2j} + (r_2)_2\mathbf{J}_2) \end{aligned} \tag{1.21}$$

Given the last equation, we get the recurrence relation:

$$r_k = Dual(\frac{\partial f}{\partial s_k}, \frac{\partial^2 f}{\partial s_k \partial x_j}) \tag{1.22}$$

1.7.3 Reverse on forward

First we set up the forward pass, given $s_1 = Dual(f_1(x), \frac{\partial f_1}{\partial x_i})$ (notice the difference with the previous modes ($j \to i$)). Then we compute:

1. Compute $s_2 = f_2(s_1) = Dual(f_2(s_1), \mathbf{J}_2 \frac{\partial f_1}{\partial x_i})$
2. The reverse mode computes the local Jacobian

$$\frac{\partial s_2}{\partial s_1} = \begin{bmatrix} \mathbf{J}_2 & 0 \\ \mathbf{H}_{2i} & \mathbf{J}_2 \end{bmatrix} \tag{1.23}$$

Then the backward pass, which gives us:

$$\begin{cases} (r_1)_1 = (r_2)_1\mathbf{J}_2 + (r_2)_2\mathbf{H}_{2i} \\ (r_1)_2 = (r_2)_2\mathbf{J}_2 \end{cases} \tag{1.24}$$

Which gives us the solution of recurrence equation:

$$r_k = Dual(\frac{\partial f}{\partial s_k}, \frac{\partial^2 f}{\partial s_k \partial x_j}) \tag{1.25}$$

1.7.4 Reverse on reverse

First we need to set up the forward pass, so we have $s_2 = f_2(s_1)$, again we need to compute its jacobian $\mathbf{J}_2 = \frac{\partial s_2}{\partial s_1}$. Then we need compute the backward pass with $r_1 = r_2\mathbf{J}_2$. Then we need to compute again the backward pass, in order to get the second order derivative. Let $\dot{r}_k$ be the second order reverse tangent for r_k, then we have:

$$\begin{aligned} \dot{r}_2 &= \mathbf{J}_2\dot{r}_1 \\ \dot{s}_1 &= (r_2\partial^2 f_2(s_1))\dot{r}_1 + \dot{s}_2\mathbf{J}_2 \end{aligned} \tag{1.26}$$

So we can get the solution of the recurrence relation with $\dot{s}_0 = e_i$:

$$\begin{cases} r_k = \mathbf{J}_K \dots \mathbf{J}_{k+1} \\ \dot{r}_k = \mathbf{J}_k \dots \mathbf{J}_1 e_i \\ (r_k\partial^2 f_k)\dot{r}_{k-1} = r_k(\partial^2 f_k \dot{r}_{k-1}) \\ \qquad\qquad\qquad = r_k\mathbf{H}_{ki} \\ \dot{s}_k = \sum_{k=1}^{K} r_k\mathbf{H}_{ki}\mathbf{J}_{k-1}\dots\mathbf{J}_1 \end{cases} \tag{1.27}$$

1.8 Implicit differentiation

1.9 Sparse AD

Often in practice, the Jacobian matrix is sparse, meaning that many of its entries are zero. In such cases, we can use sparse automatic differentiation techniques to exploit this sparsity and reduce the computational cost of computing the Hessian. For this section, we will suppose that we know the sparsity pattern of the Jacobian matrix. Consider for example the following Jacobian matrix:

$$J = \begin{bmatrix} 0 & * & 0 & * & 0 \\ 0 & 0 & 0 & * & * \\ 0 & * & * & 0 & 0 \\ * & 0 & * & 0 & 0 \end{bmatrix} \tag{1.28}$$

If we computed the Hessian using the standard AD techniques, we would need to compute 5 JVP or 4 VJP. But what if we could group some columns or rows together to reduce the number of passes needed ? Yes we can combine the columns or rows that do not share any non-zero entry. For example, with the Jacobian matrix above, we can compute the Hessian with only 2 JVP or 2 VJP. the intuition behind this comes from the fact that when we compute a JVP or a VJP, the zero entries do not contribute to the result, so we can group together the columns or rows that do not share any non-zero entry. For example, let us compute the two firsts JVPs:

$$J\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ * \end{bmatrix} \qquad \& \qquad J\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} * \\ 0 \\ * \\ 0 \end{bmatrix} \tag{1.29}$$

Combining these two JVPs, we get:

$$J \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} * \\ 0 \\ * \\ * \end{bmatrix} \tag{1.30}$$

And because we know the sparsity pattern we can then put the results back in the right place, this is called the decompression. So to compute the Hessian, we combine the following columns and so use those vectors for the JVPs:

$$\begin{bmatrix} 0 & * & 0 & * & 0 \\ 0 & 0 & 0 & * & * \\ 0 & * & * & 0 & 0 \\ * & 0 & * & 0 & 0 \end{bmatrix} \quad \Rightarrow \quad v_1 = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \; \& \quad v_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{1.31}$$

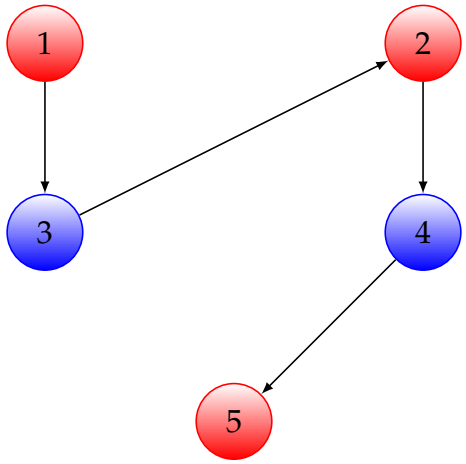
Or for the VJP:

$$\begin{bmatrix} 0 & * & 0 & * & 0 \\ 0 & 0 & 0 & * & * \\ 0 & * & * & 0 & 0 \\ * & 0 & * & 0 & 0 \end{bmatrix} \quad \Rightarrow \quad v_1 = \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix} \; \& \quad v_2 = \begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix} \tag{1.32}$$

1.9.1 Sparsity detection

The problem of grouping the columns or rows of a sparse matrix can be seen as a graph coloring problem. There is multiple way to model this problem as a graph coloring problem. For example, we can create a graph where each column is a node and there is an edge between two nodes if they share a non-zero entry. Then we can use a graph coloring algorithm to color the graph such that no two adjacent nodes have the same color. Each color then 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:

$$J = \begin{bmatrix} 0 & * & 0 & * & 0 \\ 0 & 0 & 0 & * & * \\ 0 & * & * & 0 & 0 \\ * & 0 & * & 0 & 0 \end{bmatrix} \tag{1.33}$$



Neural networks

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

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

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

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

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

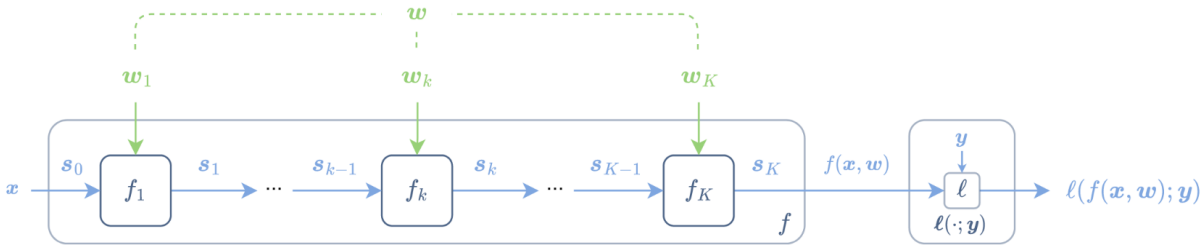


Figure 2.1: Neural network forward pass

And the backward pass can be represented like this:

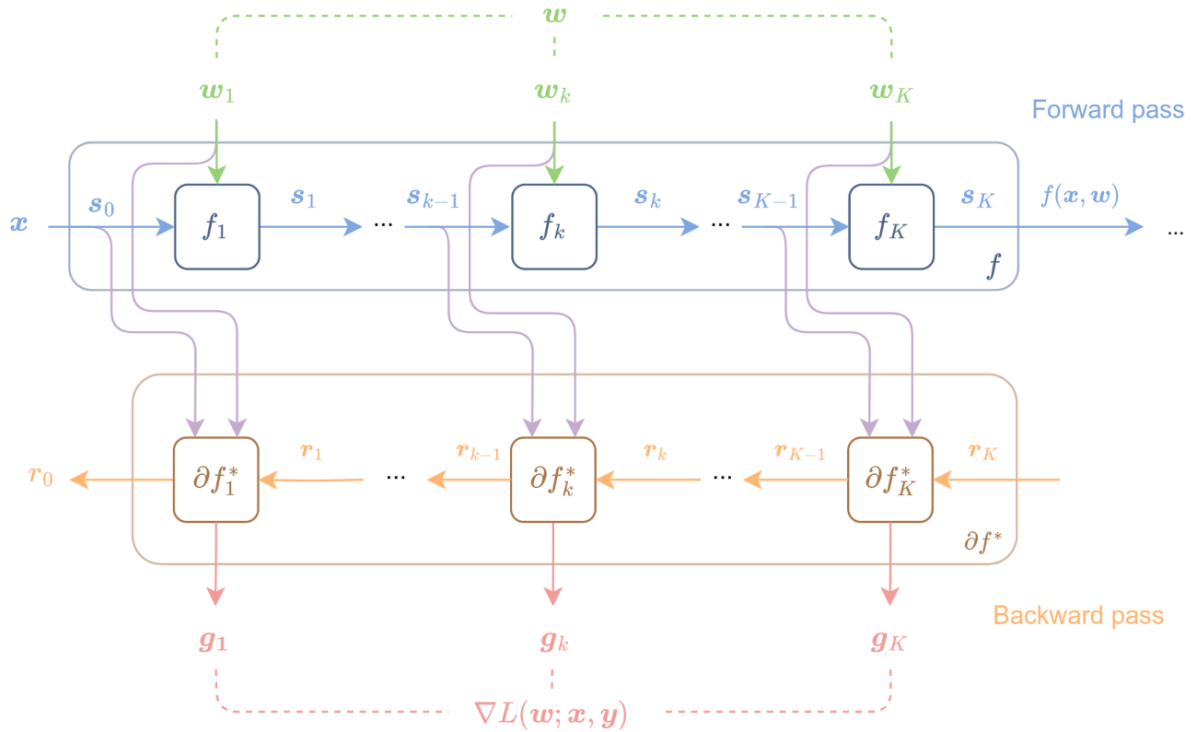


Figure 2.2: Neural network backward pass

2.1 Autoregressive models

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

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

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

p:

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

2.2 Tokenization

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

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

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

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

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

2.3 Embedding

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

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

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

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

If we choose wisely d_{emb}, which means much smaller than n_{voc}, then it is faster to compute $W_1(Cx_{-1})$ than in the previous model. Moreover we are forcing W_1C to be low-rank which can help to reduce overfitting but reduce the expressivness (capacity to capture a range of possible relation between input and output) of the model. It is useful to share the embedding between different input and output when $n_{ctx} > 1$, we also found that forcing $D = C^T$ appears to work well in practice.

2.3.1 Shared embedding

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

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

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

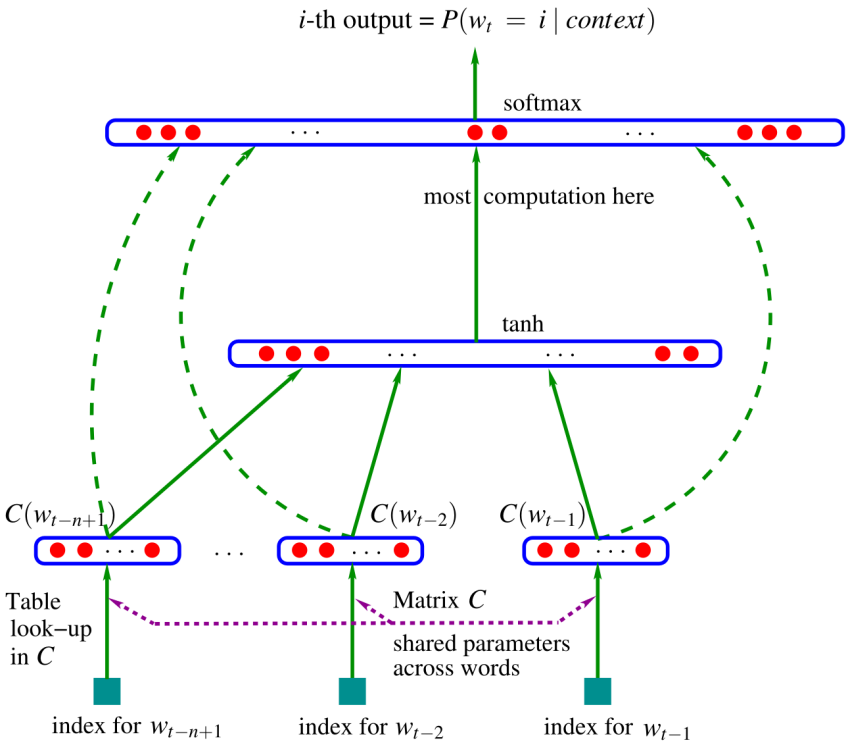


Figure 2.3: Neural network with shared embedding

2.4 Recurrent neural networks (RNN)

A RNN is a type of neural network that is designed to process sequential data by taking advantages of the memory of previous inputs. The idea is to have a hidden state h_t that

depends on the current input x_t and the previous hidden state h_{t-1}. With this idea, the network reuses the same weights at every time step and thus gives meaning to the sequence. The equations of a simple RNN are:

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

Where W, U, V are weight matrices and b, c are bias vectors.
This model presents some limitations, for example it is difficult to learn long-term dependencies because of the vanishing/exploding gradient problem. The fact that it processes a sequence step by step makes it difficult to parallelize the computations. The fact that it need to store the hidden state at each time step can be also memory-intensive for long sequences.

2.5 Attention head

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

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

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

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

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

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

2.5.1 Masked attention

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

preventing them from contributing to the attention output.
So with masked attention, the equation becomes:

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

2.5.2 Multi-head attention

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

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

2.6 Decoder-only transformer

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

2.6.1 Positional encoding

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

index i and embedding dimension d_{emb} could be defined as:

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

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

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

2.6.2 Residual connection

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

2.6.3 Layer normalization

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

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

With γ and β two learnable parameters, μ_i the mean of the elements of y_i and σ_i the standard deviation of the elements of y_i. Then we can add this in the transformer architecture like this:

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

2.6.4 Feed-forward network (FFN)

The feed-forward network consists in doing this operation:

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

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

2.6.5 Transformers variation

To investigate ?

2.7 Performances of transformers

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

- n_{ctx}: context size (number of tokens in the input sequence)
- n_{voc}: vocabulary size (number of unique tokens)
- d_{emb}: embedding size (dimension of embeddings)
- d_k, d_v: key and value dimension (per head)
- d_{ff}: feed-forward network dimension
- N: number of layers

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

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

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

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

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

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

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

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

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

Diffusion Models

Kernels

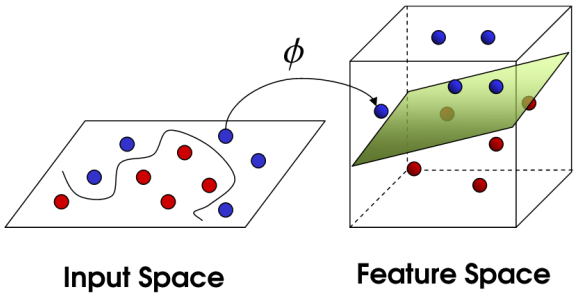


Figure 4.1: Illustration of kernels

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

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

where the kernel function then is

$$\phi(x) = \begin{pmatrix} x_1^2 \\ x_2^2 \\ \sqrt{2}x_1x_2 \end{pmatrix} \tag{4.2}$$

4.1 Reminders on scalar product

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

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

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

4.1.1 Equivalence

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

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

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

4.1.2 Hilbert space

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

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

4.2 Kernel methods for finite sets

4.2.1 Example – word embedding

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

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

4.2.4 Advantages

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

4.3 Kernels methods for continuous sets

4.3.1 Reproducibility Kernel Hilbert Space

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

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

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

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

For a RKHS H, we verify

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

The reproducing kernel property is

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

4.3.2 Properties

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

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

4.4 Polynomial kernels

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

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

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

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

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