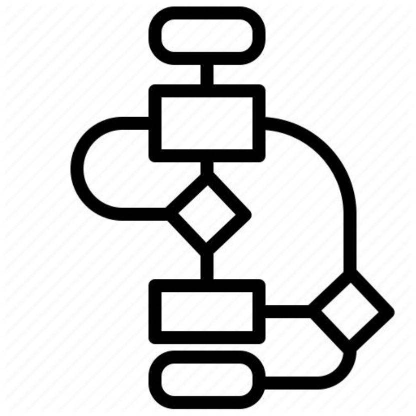


LINMA2472 - Algorithm in data science

Simon Desmidt

Academic year 2025-2026 - Q1

UCLouvain

Contents

1 Automatic differentiation **2**
1.1 Chain rule 2
1.2 Forward differentiation 2
1.3 Backward differentiation 3
1.4 Computational graph and multivariate differentiation 4
1.5 Jacobian computation 5
1.6 Memory usage 5
1.7 new part idk 7
1.8 Second order AD 7

2 Neural networks **10**
2.1 Autoregressive models 11
2.2 Tokenization 12
2.3 Embedding 12
2.4 Recurrent neural networks (RNN) 13
2.5 Attention head 14

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:

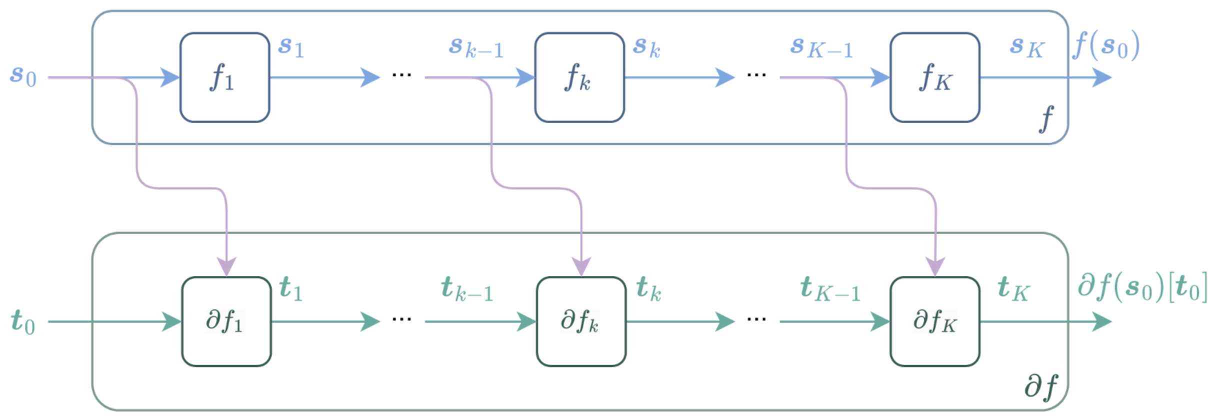


Figure 1.1: Forward differentiation

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:

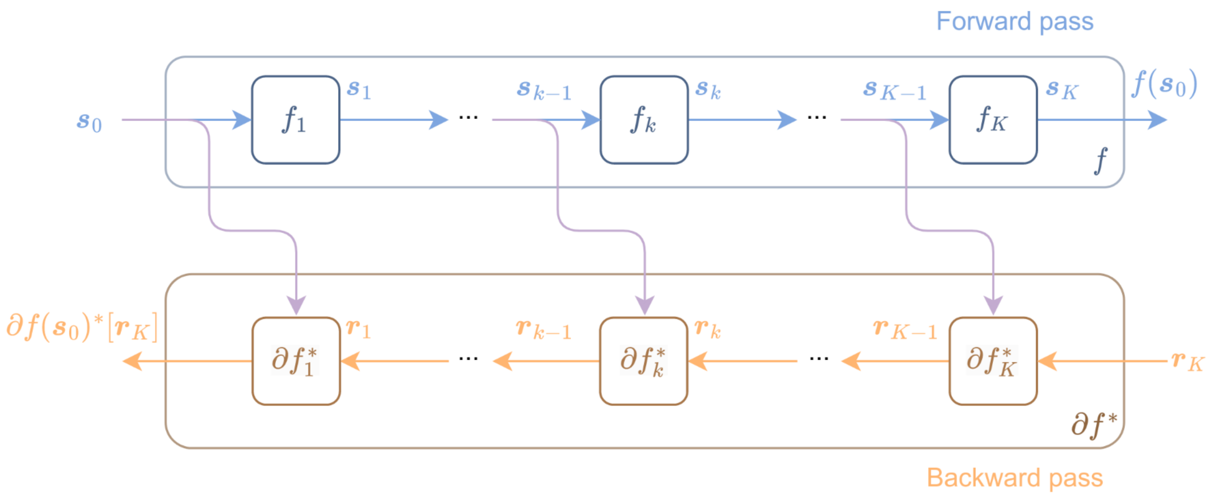


Figure 1.2: Backward differentiation

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.

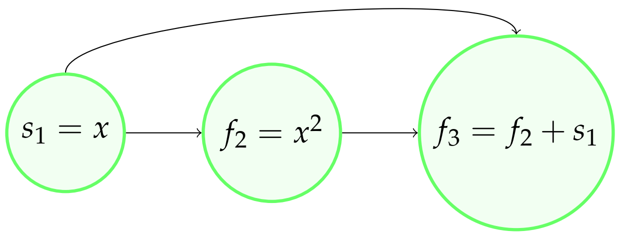
1.4 Computational graph and multivariate differentiation

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.

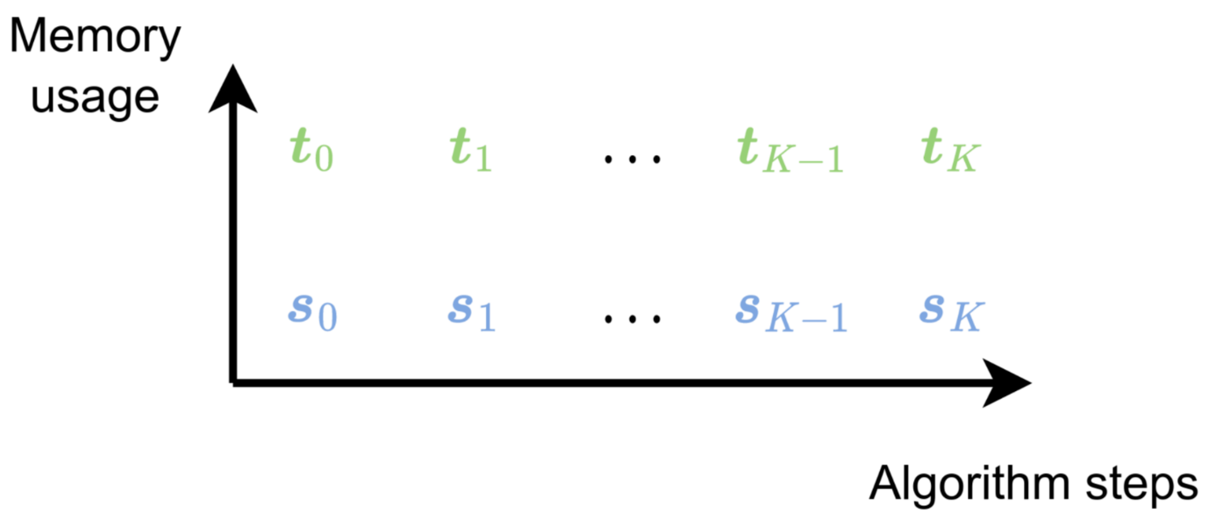


Figure 1.3: Forward mode memory usage

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.

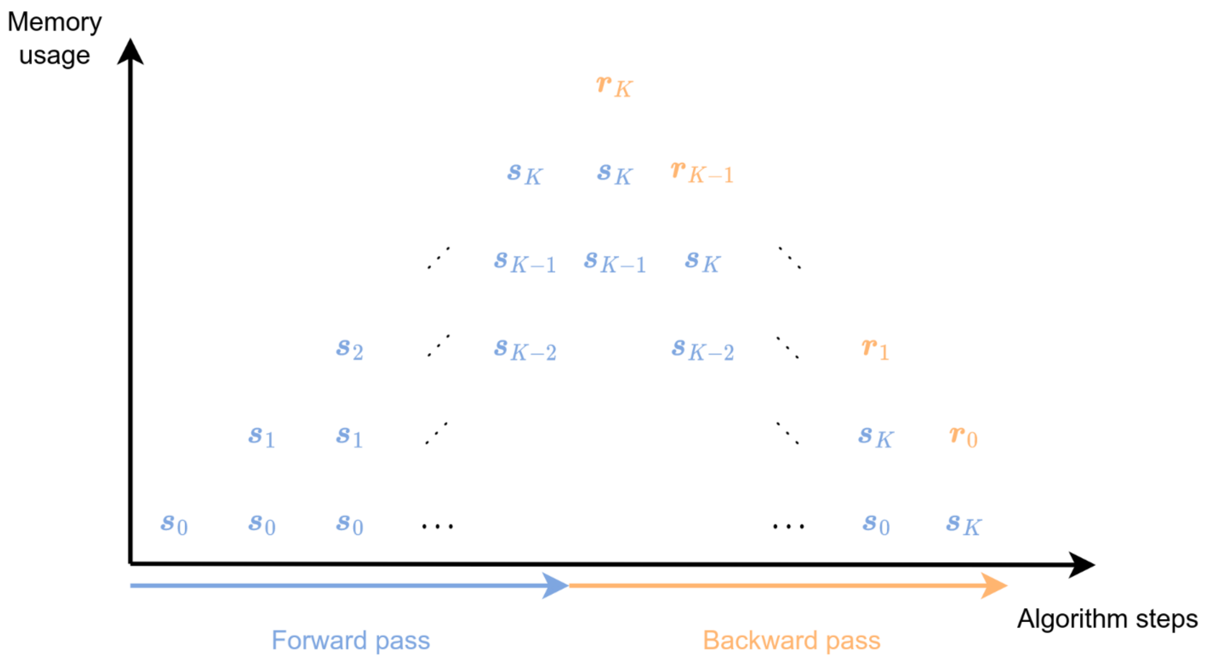


Figure 1.4: Backward mode memory usage

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 new part idk

1.8 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.

1.8.1 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 $g_k(x) = f_k \circ \cdots \circ f_1$:

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

1.8.2 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:

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

1.8.3 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:

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

1.8.4 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 $\dot{s}_0 = e_i$:

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

Neural networks

TODO link tangent with 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:

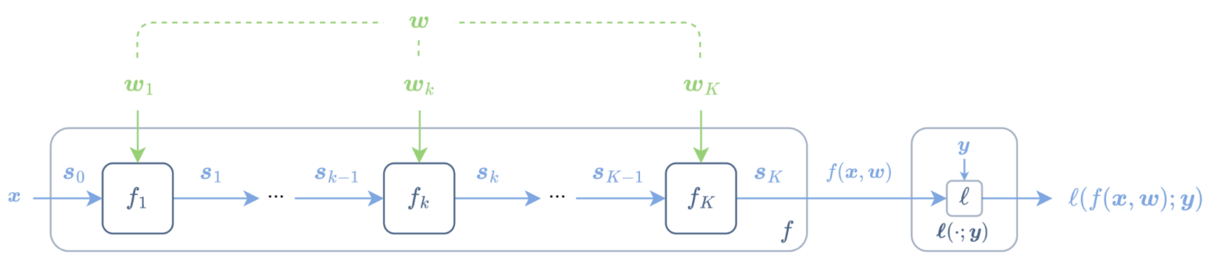


Figure 2.1: Neural network forward pass

And the backward pass can be represented like this:

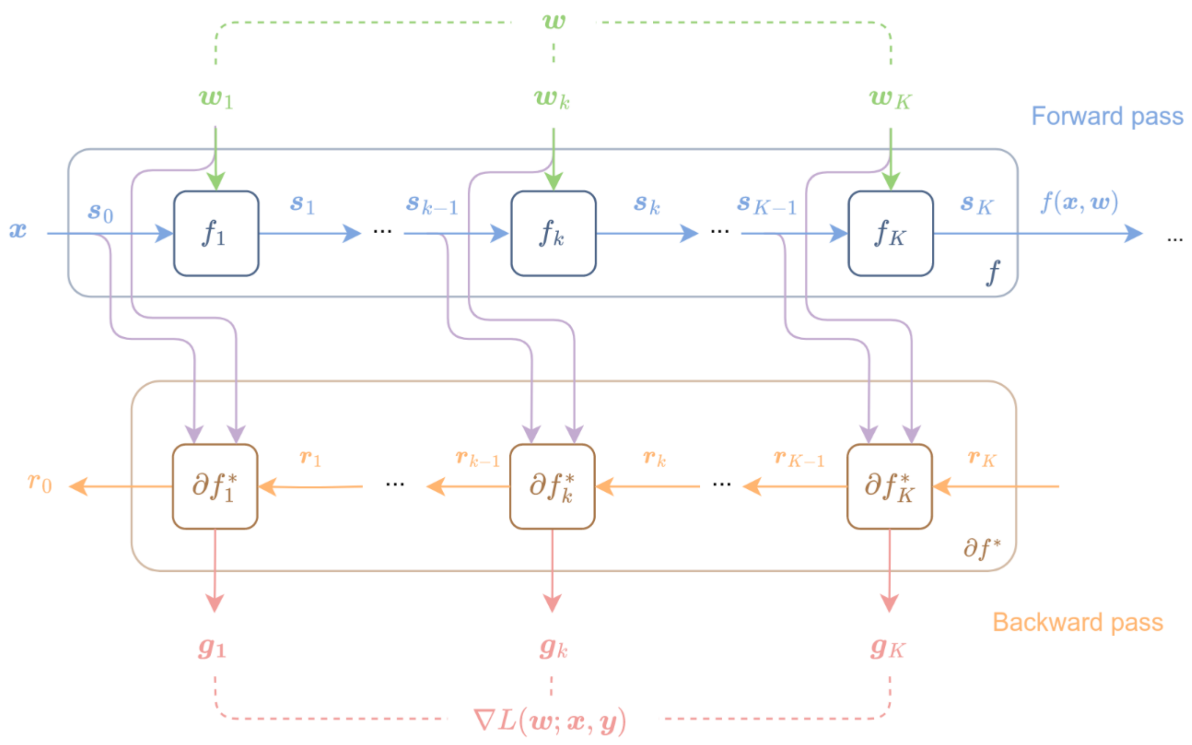


Figure 2.2: Neural network backward pass

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 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:

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

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

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

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

2.3 Embedding

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

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

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:

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

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:

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

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

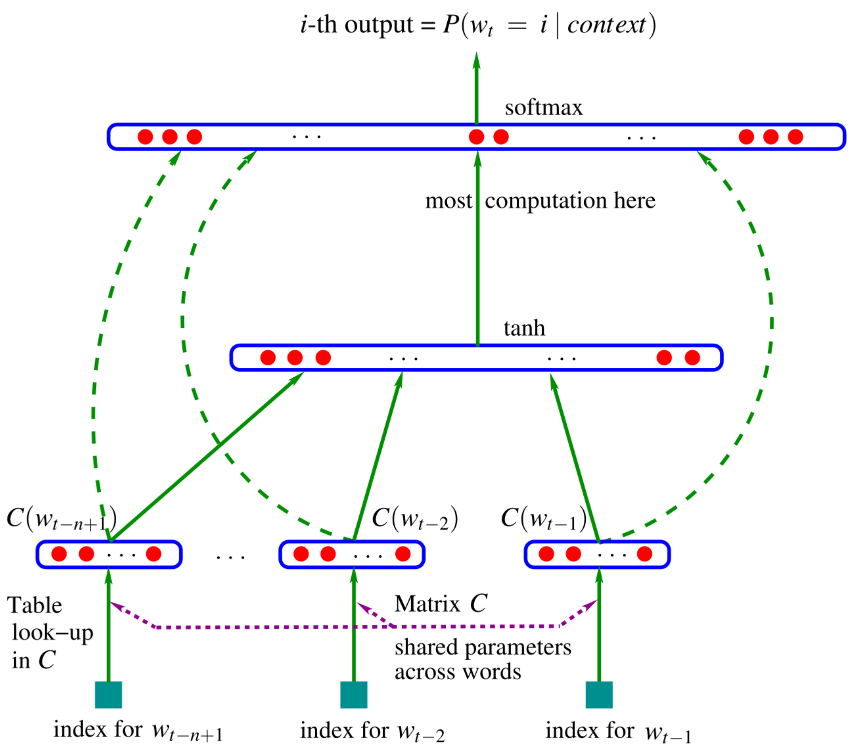


Figure 2.3: Neural network with shared embedding

2.4 Recurrent neural networks (RNN)

TODO

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:

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

where $\alpha = \text{softmax}(< q, k_1 >, \cdots, < q, k_{n_{ctx}} >)$ is the attention weight for key k_i.