An introduction to Neural networks

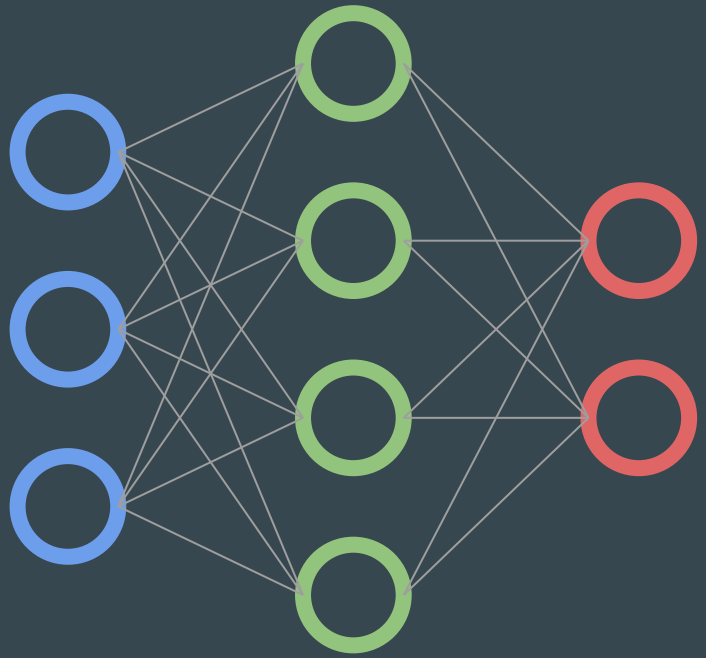
By Goll Sébastien

AI, ML, NN, DL, RL, WTF ?

People in machine learning LOVE acronyms, if you don't remember some, ask.

- **AI: Artificial Intelligence** : Set of techniques that allows machines to mimic intelligence in some aspect (ex: A*)
- **ML: Machine Learning**: Subset of AI that let the model/algorithm learn by itself (ex: K-Means)
- **NN: Neural Network**: Type of ML model which uses neurons like connection and structures (ex: Perceptron)
- **DL: Deep Learning**: Subset of ML using deep NN (ex: RNN, CNN...)
- **RL: Reinforcement Learning**: Subset of ML that allows the model to learn from a reward

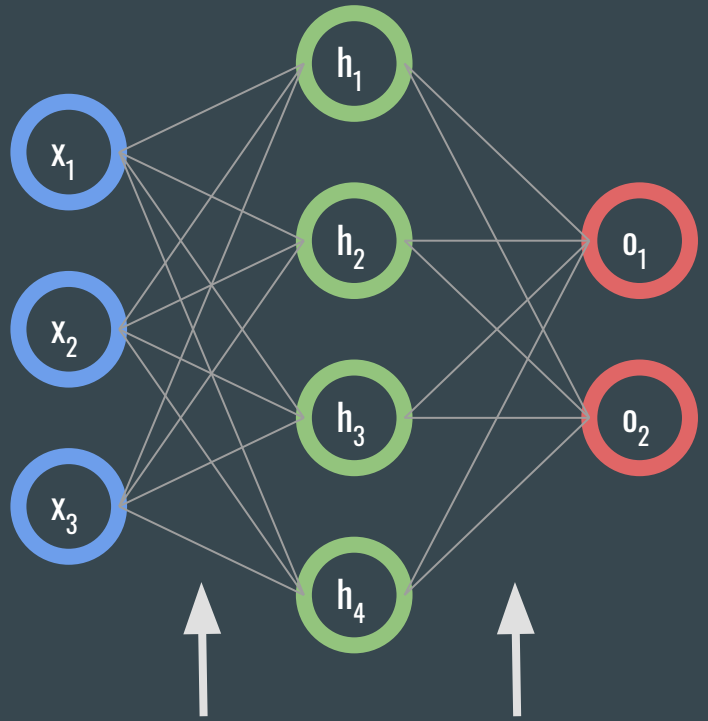
What is a NN ?



Input

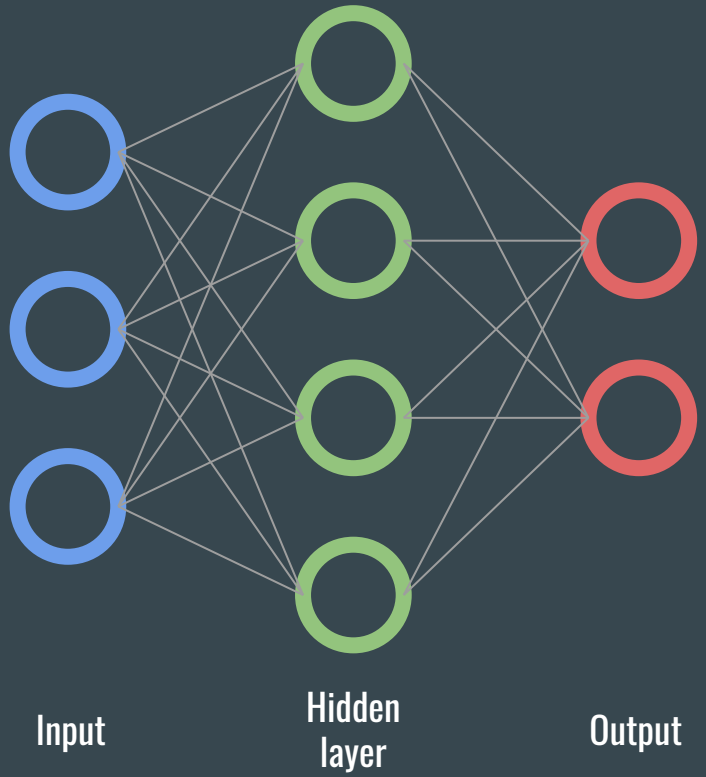
Hidden layer

Output

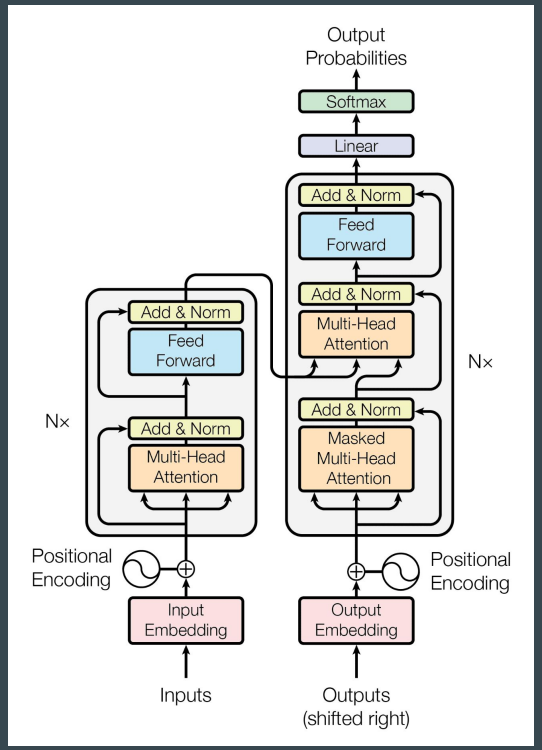


Matrix multiplication to switch between layers

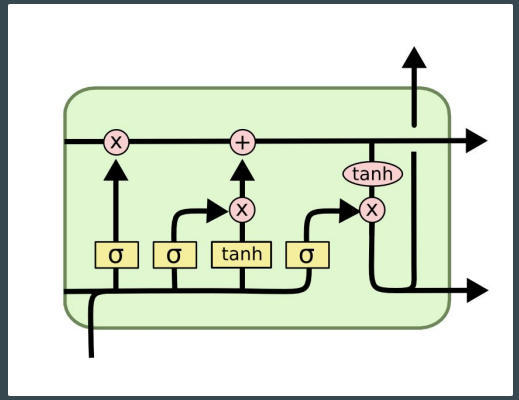
Multiple representations to fit your needs



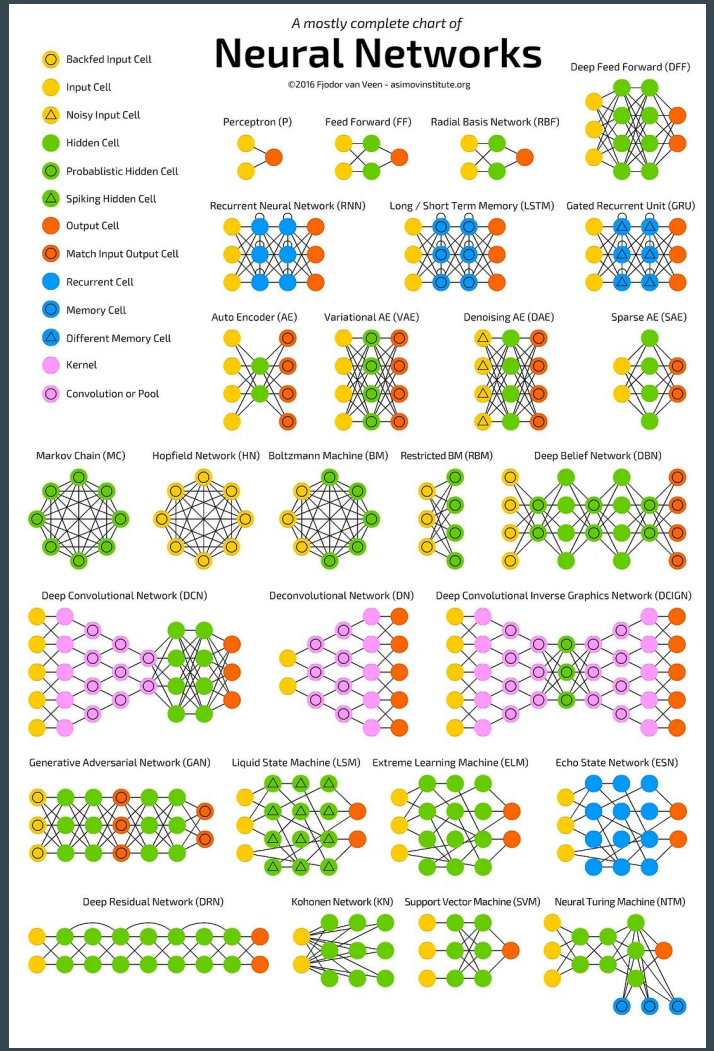
Complexe behaviors emerge



Transformer



LSTM



NN, how it works ?

Most NN works in two main steps:

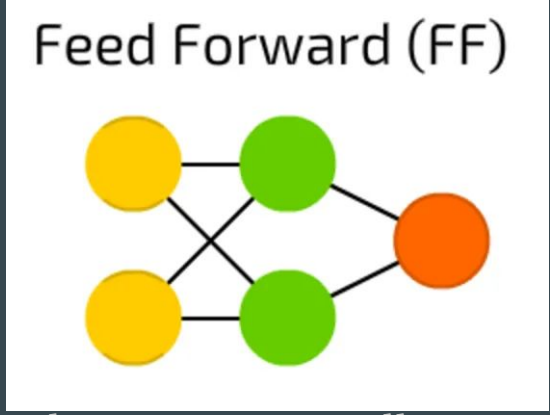
- Forward propagation

  This takes the input, and calculate the output by multiplying matrices

- Backward propagation

  From the output given by the forward propagation, and the target result, we can propagate information backward into the network to update the values of the matrices

Forward propagation



General formula : $h_i = g(h_{i-1} W_i + b_i)$

h_i: the output of the current layer (it might be the final output, in that case we usually write it as $\hat{y}$ or sometime y)

h_{i-1}: the output of the previous layer (it might be the input, in that case we write it as x)

W_i: the weight matrix for switching from layer i-1 to layer i, the dimension of this matrix depend on the dimensions of the input and the output of the layer

b_i: the bias, a constant added to the result of the matrix multiplication (it is a vector)

g: the activation function, it is a non-linear function that transform the initial matrix computation

Activation function ? Which and why ?

Why non linear (not ax+b)

-> The function composition (g(f(x))) of 2 linear function give a linear function

-> All the layers could be simplified with **one** layer

Which function to choose ?

- Sigmoid -> $g(z) = \frac{1}{1 + e^{-z}}$
- Tanh -> $\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$
- ReLu -> $g(z) = \max(0,z)$

In an hidden layer, we usually never use the sigmoid function, tanh is always better.

ReLu is not as good as tanh, but it is easier to work with

For the last layer, it's different

For the last layer, it's a little bit different.

We want the function to match the task that our model has.

Choose between 3 functions for the 3 main tasks that the models can do:

- Sigmoid : when we want a binary classification (true or false, or between 0 and 1) (ex: is this text in french)
- Identity : when we want a real number (ex: given today weather, what will be the wind speed tomorrow)
- Softmax: when we want multiclass classification (ex: is this image a car, a boat, a plane...) the sum of all the values is 1, it is a probability distribution between **n** classes

$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g^{[2]}(z) = z$$

$$\text{Softmax}(z_j) = \frac{e^{z_j}}{\sum_{c=1}^{C} e^{z_c}}$$

Backpropagation, the loss function

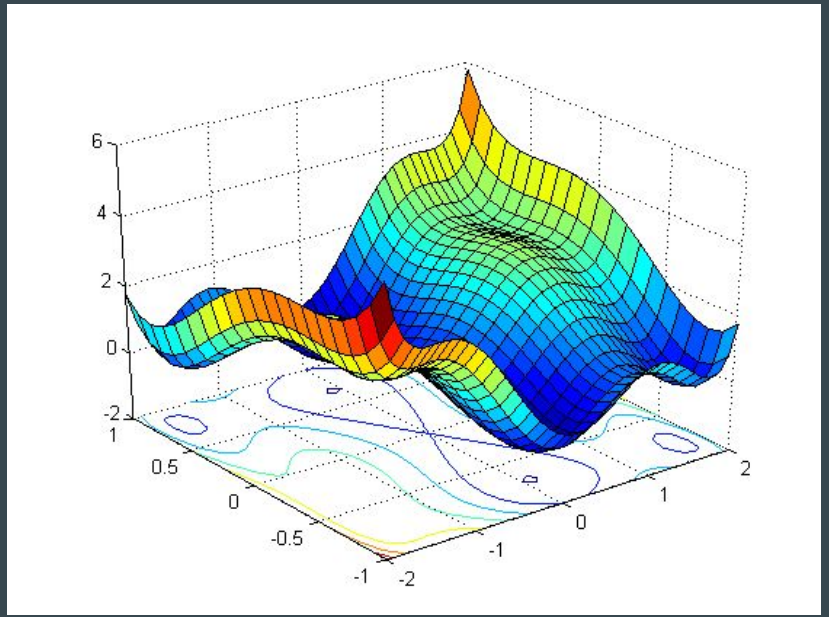
We want to changes the parameters (**weights** and **biases** of the model) so that it's performances are better.

To do that we introduce the notion of loss function, they quantify the error made by our model with its current parameters.

The loss functions depend on the task trained.

For multiclass classification, we use **Cross entropy**.

$$L(\hat{y}, y) = -\sum_{c=1}^{C} y_c \log(\hat{y}_c)$$



Backpropagation, using gradients

We want to minimise the loss value given by our model, to do that we use gradients.

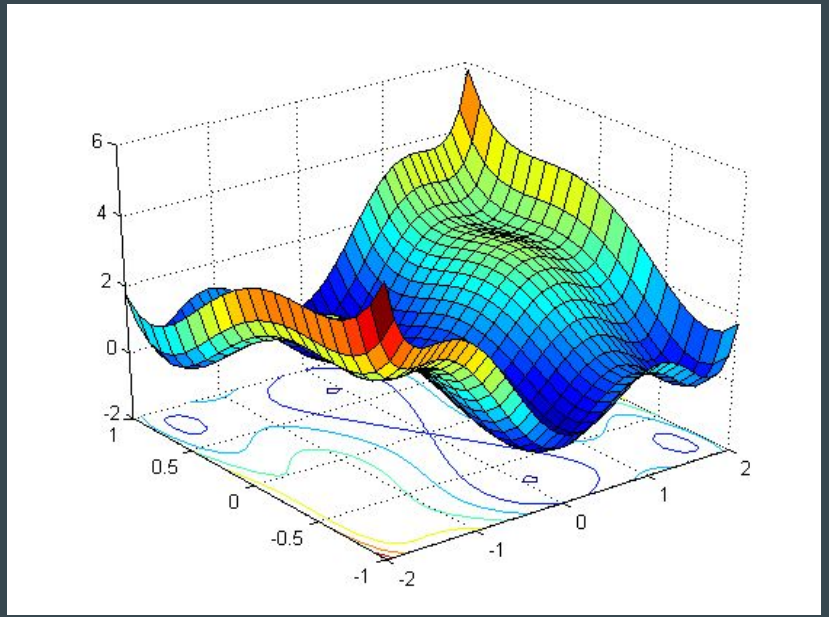
Gradients are like derivative but on higher dimensions, since we work with matrices and vectors, we cannot use a simple derivative.

We want to calculate the gradient in relation to each parameter:

$$\frac{\partial L}{\partial \theta}$$

Where $\theta \in \{W_i, b_i\}$

Given this value, we can update the parameters



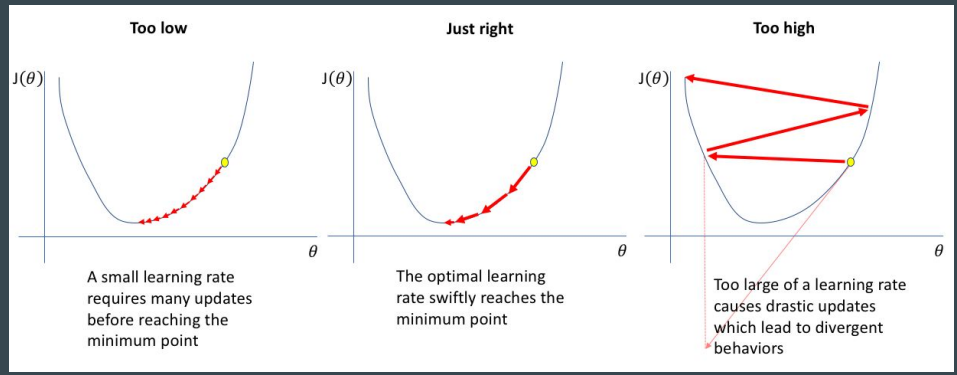
Backpropagation, formula for updating the parameters

Once the loss gradient computed, we can update the parameters like this (exemple for the first weight):

$$W^{[1]} = W^{[1]} - \alpha \frac{\partial J}{\partial W^{[1]}}$$

Here J is the loss, and α is the learning rate. (usually around 1e-4)

Learning rates must be fined tuned depending on the model, too high or too low, and the training will not be efficient.



Backpropagation, how to compute the gradient.

Good new, nowadays, most of the NN frameworks (pytorch, keras...) compute the loss for you !

Bad new, since it's an introduction, you will still have to implement it.

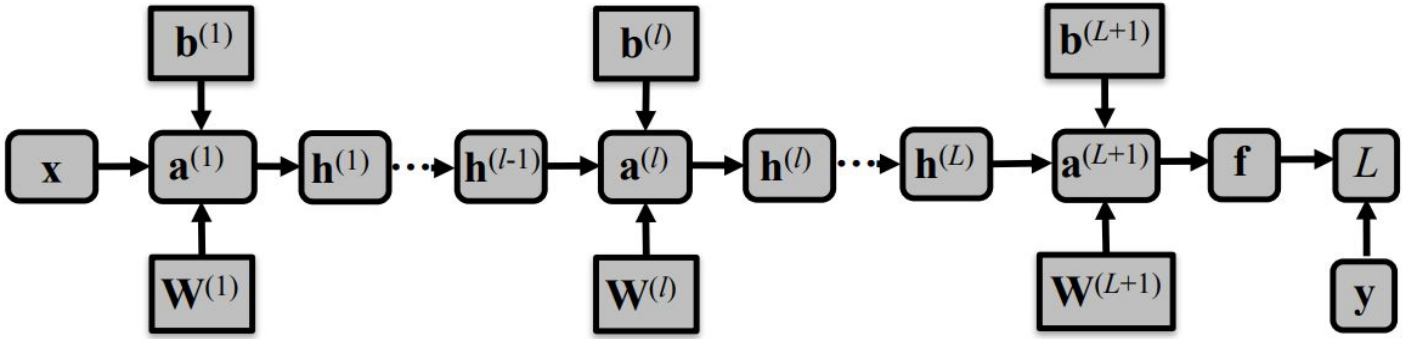
The main technique used to compute the gradient is called the chain rule: $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$

This means that we can reuse and simplify part of the computation.

We can then create a big gradient graph showing how to compute the gradient of the model

$$\mathbf{a}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \qquad \mathbf{a}^{(l)} = \mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)} \qquad \mathbf{a}^{(L+1)} = \mathbf{W}^{(L+1)}\mathbf{h}^{(L)} + \mathbf{b}^{(L+1)}$$

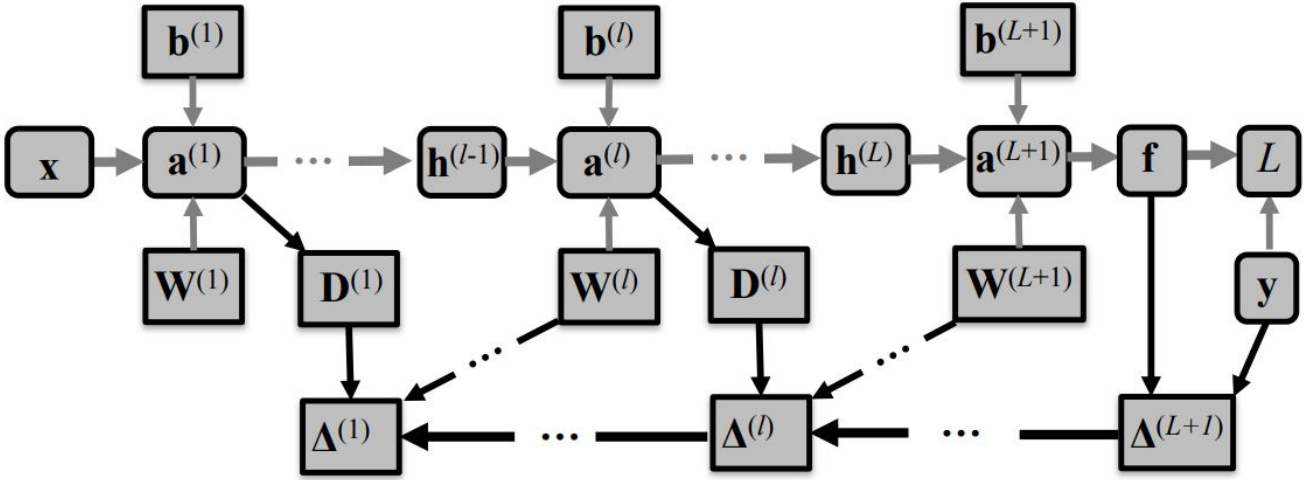
$$\mathbf{h}^{(1)} = \mathrm{act}(\mathbf{a}^{(1)}) \qquad \mathbf{h}^{(l)} = \mathrm{act}(\mathbf{a}^{(l)}) \qquad \mathbf{f} = \mathrm{out}(\mathbf{a}^{(L+1)})$$



The last layer gradient depend on the loss and the out function.

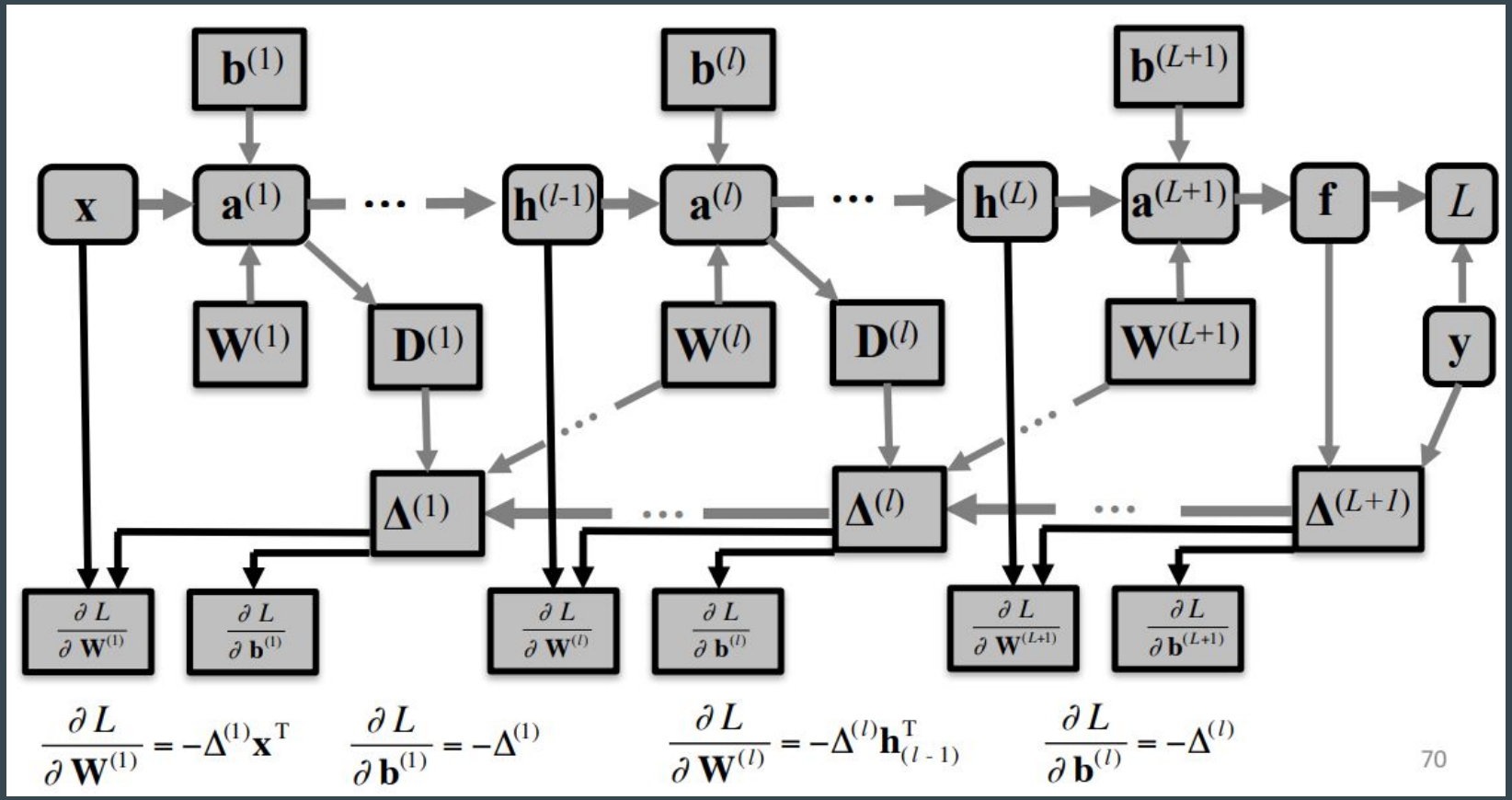
In this case, we use Cross entropy and softmax

D is the gradient of the activation function, it depend on the function



$$\Delta^{(l)} = \mathbf{D}^{(l)}\mathbf{W}^{\mathrm{T}(l+1)}\Delta^{(l+1)} \qquad \frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{a}^{(l)}} = \mathbf{D}^{(l)} \qquad \frac{\partial \mathbf{a}^{(l+1)}}{\partial \mathbf{h}^{(l)}} = \mathbf{W}^{\mathrm{T}(l+1)} \qquad \frac{\partial L}{\partial \mathbf{a}^{(L+1)}} = -\left[\mathbf{y} - \mathbf{f}(\mathbf{x})\right] = -\Delta^{(L+1)}$$

Final chain visualisation



Conclusion

What we've seen:

- Neural networks are a bunch of matrix multiplication one after the other
- The formula for computing a layer is $h_i = g(h_{i-1} W_i + b_i)$
- Given a loss function (error function), we compute the gradients to update the parameters
- The formula for updating parameters : $$W^{[1]} = W^{[1]} - \alpha \frac{\partial J}{\partial W^{[1]}}$$
- To compute the gradient, we use the chain rule, and the graph given on the previous slide

Conclusion

This is a complex subject, but far more complex in theory than in practice.

The goal is to give you a shallow understanding of the topic with the following exercice.

DO NOT HESITATE TO ASK QUESTION **DURING** THE EXERCICE

Slides by Goll Sébastien 2023

Inspired by INF8215 and INF8225 for Polytechnique Montréal