Introduction to Deep Learning

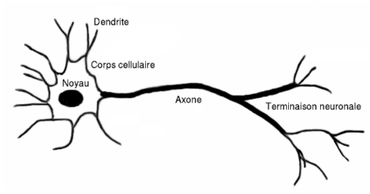
The historical multilayer perceptron

J. Rynkiewicz

Université Paris 1

2022

The simple perceptron



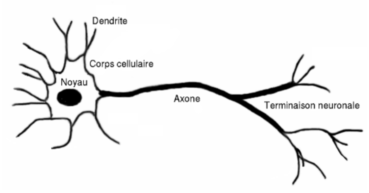
Originating from the analogy with biology, the formal neuron was introduced by McCullogh and Pitts in 1943. It is defined as follows :

- Real inputs $x_i, i \in \{1, \cdots, d\}$
- Weights $W_i, i \in \{0, \cdots, d\}$

The weights W_0 is related to a constant input, the opposite of W_0 can be seen as a threshold value, beyond which the neuron is activated.
The neuron performs the following two operations by computing :

1. Its potential : $W_0 + \sum_{i=1}^{d} W_i x_i$

The simple perceptron



Originating from the analogy with biology, the formal neuron was introduced by McCullogh and Pitts in 1943. It is defined as follows :

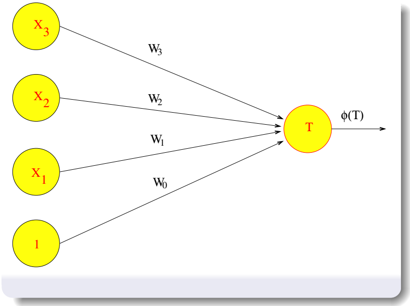
- Real inputs $x_i, i \in \{1, \cdots, d\}$
- Weights $W_i, i \in \{0, \cdots, d\}$

The weights W_0 is related to a constant input, the opposite of W_0 can be seen as a threshold value, beyond which the neuron is activated.
The neuron performs the following two operations by computing :

1. Its potential : $W_0 + \sum_{i=1}^{d} W_i x_i$
2. Its activation, thanks to an activation function ϕ : $\phi\left(W_0 + \sum_{i=1}^{d} W_i x_i\right)$

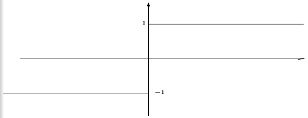
The formal neuron

The weights W_i represent the synaptic weights, $T := W_0 + \sum_{i=1}^{d} W_i x_i$ is the potential and $F_W(x) := \phi(T)$ the output of the axon. The activation functions $\phi : \mathbb{R} \longrightarrow \mathbb{R}$ are generally non-linear, for example the sign function or the family of sigmoid functions.



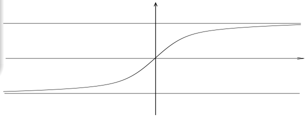
Sign function

$$\begin{cases} \phi(x) = 1 \text{ si } x \geq 0 \\ \phi(x) = -1 \text{ si } x < 0 \end{cases}$$



Sigmoid function

$$\phi(x) = c\frac{e^{kx}-1}{e^{kx}+1} + r;\ c, k > 0$$



Linear classificiation

Let $\mathcal{S}^+$ and $\mathcal{S}^-$ be two subsets of $\mathbb{R}^d$. These two sets are linearly separable if and only if, $W = (W_0, \cdots, W_d) \in \mathbb{R}^{d+1}$ exists such that :

$$\forall x \in \mathcal{S}^+ : W_0 + \sum_{i=1}^d W_i x_i > 0$$
$$\forall x \in \mathcal{S}^- : W_0 + \sum_{i=1}^d W_i x_i < 0$$

A perceptron F_W with sign function for activation function can separate these two sets :

$$F_W(x) = 1 \text{ si } x \in \mathcal{S}^+$$
$$F_W(x) = -1 \text{ si } x \in \mathcal{S}^-$$

Learning of the perceptron

1. We initialize the weights randomly
2. At time t a vector x is presented, let $\varepsilon > 0$.
   - If $F_W(x) = 1$ instead of -1 :
     $$\forall i \in \{1, \cdots, d\} : W_i(t+1) = W_i(t) + \varepsilon x_i$$
   - If $F_W(x) = 1$ instead of -1 :
     $$\forall i \in \{1, \cdots, d\} : W_i(t+1) = W_i(t) - \varepsilon x_i$$
   - If $F_W(x)$ gives the right answer, the wheights are unchanged.

Linear classificiation

Let $\mathcal{S}^+$ and $\mathcal{S}^-$ be two subsets of $\mathbb{R}^d$. These two sets are linearly separable if and only if, $W = (W_0, \cdots, W_d) \in \mathbb{R}^{d+1}$ exists such that :

$$\begin{array}{l} \forall x \in \mathcal{S}^+ : W_0 + \sum_{i=1}^{d} W_i x_i > 0 \\ \forall x \in \mathcal{S}^- : W_0 + \sum_{i=1}^{d} W_i x_i < 0 \end{array}$$

A perceptron F_W with sign function for activation function can separate these two sets :

$$\begin{array}{l} F_W(x) = 1 \text{ si } x \in \mathcal{S}^+ \\ F_W(x) = -1 \text{ si } x \in \mathcal{S}^- \end{array}$$

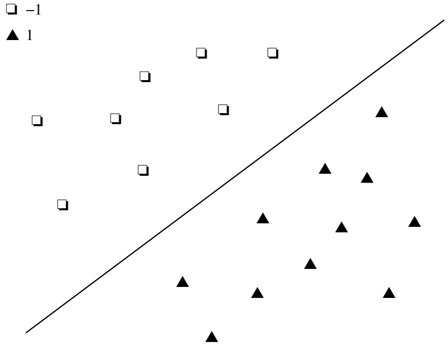
Learning of the perceptron

1. We initialize the weights randomly
2. At time t a vector x is presented, let $\varepsilon > 0$.
   - If $F_W(x) = 1$ instead of -1 :
     $$\forall i \in \{1, \cdots, d\} : W_i(t+1) = W_i(t) + \varepsilon x_i$$
   - If $F_W(x) = 1$ instead of -1 :
     $$\forall i \in \{1, \cdots, d\} : W_i(t+1) = W_i(t) - \varepsilon x_i$$
   - If $F_W(x)$ gives the right answer, the wheights are unchanged.

The weight vector of the perceptron will converge after a finite number of iterations.

Linearly separable problem

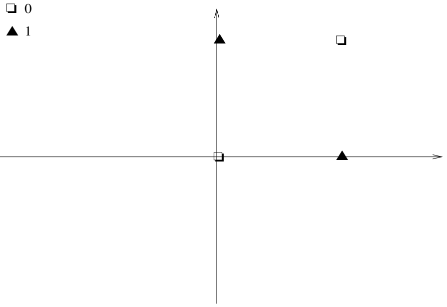
The perceptron will find one of the lines that separates the two sets to be classified.



Limitation of the simple perceptron

The perceptron cannot solve non-linear problems :

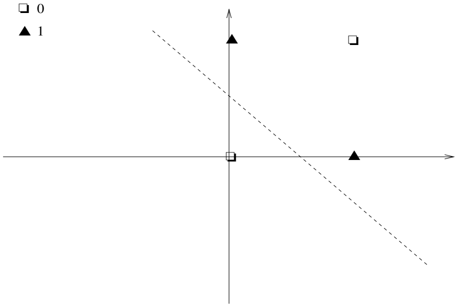
Le problème XOR



Limitation of the simple perceptron

The perceptron cannot solve non-linear problems :

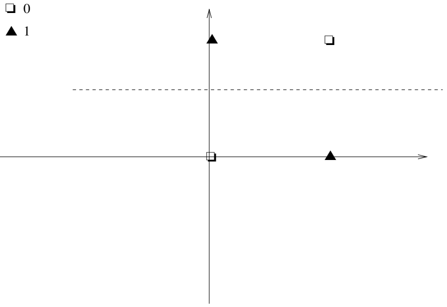
Le problème XOR



Limitation of the simple perceptron

The perceptron cannot solve non-linear problems :

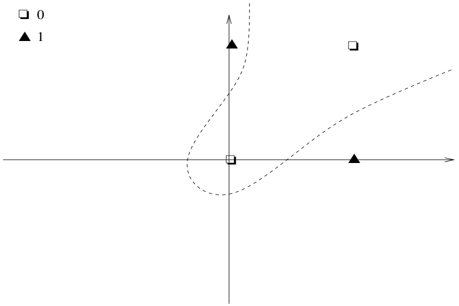
Le problème XOR



Limitation of the simple perceptron

The perceptron cannot solve non-linear problems :

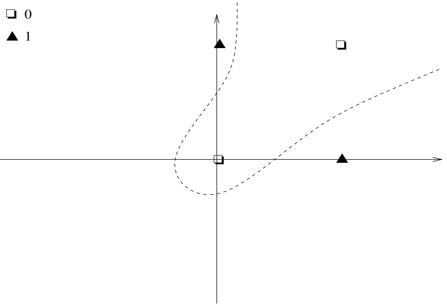
Le problème XOR



Limitation of the simple perceptron

The perceptron cannot solve non-linear problems :

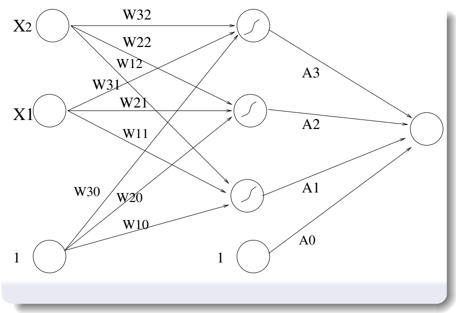
Le problème XOR



Note : We can manage to separate this set linearly if we work in the superset :$\{x, y, xy\}$

$$x + y - 0.5 - 2xy > 0 \text{ si } (x, y) \in \{(0, 1), (1, 0)\}$$
$$x + y - 0.5 - 2xy < 0 \text{ si } (x, y) \in \{(0, 0), (1, 1)\}$$

The multilayer perceptron (MLP)



$$f_\theta(x_1, x_2) = a_0 \\ +a_1\phi(w_{10} + x_1 \times w_{11} + x_2 \times w_{21}) \\ +a_2\phi(w_{20} + x_1 \times w_{21} + x_2 \times w_{22}) \\ +a_3\phi(w_{30} + x_1 \times w_{31} + x_2 \times w_{32})$$

where ϕ is the activation function.
In the sequel, $\theta = (w_{10}, \cdots, a_N)$ will be the parameter vector of the MLP.

Théorème

Let f_θ be a multilayer perceptron with one hidden layer, where ϕ is a strictly increasing bounded function. Let K be a compact subset of $\mathbb{R}^m$.
Then, for all function f continuous with compact support ($f \in C(K)$), $f : \mathbb{R}^m \longrightarrow \mathbb{R}^s$ and for $\epsilon > 0$, an MLP with N hidden units exists with parameter vector θ, such that $\forall (x_1, \cdots, x_m) \in \mathbb{R}^m$

$$\|f(x_1, \cdots, x_m) - f_\theta(x_1, \cdots, x_m)\| < \epsilon$$

Estimation of MLP's weights

We have a sequence of observations $(x_t, y_t)_{1\leq t\leq n}$. x_t are the explanatory variables and y_t the variables to be explained. Estimation amounts to minimizing the empirical mean of a cost function in θ :
$C_n(\theta) = \frac{1}{n}\sum_{t=1}^{n} e(f_\theta(x_t), y_t) := \frac{1}{n}\sum_{t=1}^{n} e_t$
Cost functions :

1. Régression
   - If $y_t \in \mathbb{R}$, $e_t = e(f_\theta(x_t), y_t) = (f_\theta(x_t) - y_t))^2$.
   - If $y_t \in \mathbb{R}^s$, $e_t = e(f_\theta(x_t), y_t) = \|f_\theta(x_t) - y_t)\|^2$, the Euclidean norm.

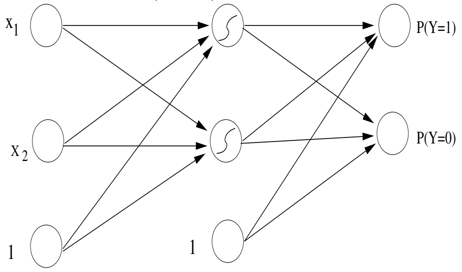
Estimation of MLP's weights

We have a sequence of observations $(x_t, y_t)_{1\leq t\leq n}$. x_t are the explanatory variables and y_t the variables to be explained. Estimation amounts to minimizing the empirical mean of a cost function in θ :
$C_n(\theta) = \frac{1}{n}\sum_{t=1}^{n} e(f_\theta(x_t), y_t) := \frac{1}{n}\sum_{t=1}^{n} e_t$
Cost functions :

1. Régression
   - If $y_t \in \mathbb{R}$, $e_t = e(f_\theta(x_t), y_t) = (f_\theta(x_t) - y_t))^2$.
   - If $y_t \in \mathbb{R}^s$, $e_t = e(f_\theta(x_t), y_t) = \|f_\theta(x_t) - y_t)\|^2$, the Euclidean norm.
2. Classification
   - If $y_t \in (1, \cdots, K)$, for K classes.
     The MLP will have K outputs $\left(f_\theta^l(x_t)\right)_{1\leq l\leq K}$. We will have :
     $P_\theta(Y_t = k\,|X_t = x_t\,) = \frac{\exp\left(f_\theta^k(x_t)\right)}{\sum_{l=1}^K \exp\left(f_\theta^l(x_t)\right)}$ and the conditional log-likelihood :
     $$l_\theta(\dots) = \sum_{t=1}^{n}\sum_{k=1}^{K} \mathbf{1}_k(y_t)\log\left(P_\theta(Y_t = k\,|x_t\,)\right)$$
   - So the "cross entropy" cost function (opposite of the log-likelihood) :
     $e_t = e(f_\theta(x_t), y_t) = -\sum_{k=1}^K \left[\mathbf{1}_k(y_t)\log\frac{\exp\left(f_\theta^k(x_t)\right)}{\sum_{l=1}^K \exp\left(f_\theta^l(x_t)\right)}\right]$

Computation of the gradient

- For fixed observations $\left(\begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, \cdots, \begin{pmatrix} x_n \\ y_n \end{pmatrix}\right)$, we seek to minimize the function $\theta \mapsto C_n(\theta) = \frac{1}{n}\sum_{t=1}^{n} e(f_\theta(x_t), y_t)$, where $\theta \in \mathbb{R}^d$. We can only approximate the solution numerically. For this purpose, we compute the derivatives for $t \in \{1, \cdots, n\}$ of $\left(\frac{\partial e(f_\theta(x_t), y_t)}{\partial \theta_i}\right)_{1 \le i \le d}$ and we proceed numerically.
- To illustrate this computation, we will consider an elementary model with $x_t = \begin{pmatrix} x_{1t} \\ x_{2t} \end{pmatrix}$ et $y_t \in \{0, 1\}$:



Elementary MLP Example

The Elementary MLP : $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$, $y_t \in \{0,1\}$ and $\theta = (W^1_{10}, \cdots, W^2_{22})$:

$$e(f_\theta(x), y) = -\mathbf{1}_{\{0\}}(y)\ln\left(p_{1\theta}(x_1,x_2)\right) - \mathbf{1}_{\{1\}}(y)\ln\left(p_{2\theta}(x_1,x_2)\right)$$

with

- The first layer $z^1 = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$.
- The second layer $z^2 = \phi^2(W^1 z^1) = \begin{pmatrix} \tanh(w^1_{11}x_1 + w^1_{12}x_2 + w^1_{10}) \\ \tanh(w^1_{21}x_1 + w^1_{22}x_2 + w^1_{20}) \end{pmatrix}$
- The third layer $z^3 = \begin{pmatrix} p_1(\theta) \\ p_2(\theta) \end{pmatrix} = \phi^3(W^2 z^2) =$
$$\begin{pmatrix} \frac{\exp\left(w^2_{11}z^2_1 + w^2_{12}z^2_2 + w^2_{10}\right)}{\exp\left(w^2_{11}z^2_1 + w^2_{12}z^2_2 + w^2_{10}\right) + \exp\left(w^2_{21}z^2_1 + w^2_{22}z^2_2 + w^2_{20}\right)} \\ \frac{\exp\left(w^2_{21}z^2_1 + w^2_{22}z^2_2 + w^2_{20}\right)}{\exp\left(w^2_{11}z^2_1 + w^2_{12}z^2_2 + w^2_{10}\right) + \exp\left(w^2_{21}z^2_1 + w^2_{22}z^2_2 + w^2_{20}\right)} \end{pmatrix}$$
- The cost function $e\left(\begin{pmatrix} p_1 \\ p_2 \end{pmatrix}, y\right) = -\mathbf{1}_{\{0\}}(y)\ln(p_1) - \mathbf{1}_{\{1\}}(y)\ln(p_2)$

Computation of the derivative (1)

- For all parameter $\theta_i \in \{w^1_{10}, \cdots, w^2_{22}\}$, we can use the computation chain :

$$\frac{\partial e(f_\theta(x),y)}{\partial \theta_i} = \frac{\partial e(f_\theta(x),y)}{\partial z^3_1}\frac{\partial z^3_1}{\partial z^2_1}\frac{\partial z^2_1}{\partial \theta_i} + \frac{\partial e(f_\theta(x),y)}{\partial z^3_1}\frac{\partial z^3_1}{\partial z^2_2}\frac{\partial z^2_2}{\partial \theta_i} + \frac{\partial e(f_\theta(x),y)}{\partial z^3_2}\frac{\partial z^3_2}{\partial z^2_1}\frac{\partial z^2_1}{\partial \theta_i} + \frac{\partial e(f_\theta(x),y)}{\partial z^3_2}\frac{\partial z^3_2}{\partial z^2_2}\frac{\partial z^2_2}{\partial \theta_i}$$

  but this calculation is redundant, the colored quantities appear several times.
- We can still notice that, for the cross entropy, the derivative with respect to the output of the neural network is easily obtained :

$$\frac{\partial e(z,y)}{\partial z} = z - y,$$

  where, with a slight abuse of notation, z is the output vector of the network after application of the softmax function and y is the class encoded on the simplex (one-hot encoding).

Computation of the derivative (2)

- The back-propagation algorithm computes first and without redundancy $\delta_i^L = \frac{\partial e(f_\theta(x),y)}{\partial z_i^L}$:

$$\delta_i^L = \frac{\partial e(f_\theta(x),y)}{\partial z_i^L} = \sum_j \frac{\partial e(f_\theta(x),y)}{\partial z_j^{L+1}} \frac{\partial z_j^{L+1}}{\partial z_i^L} = \sum_j \delta_j^{L+1} \frac{\partial z_j^{L+1}}{\partial z_i^L}$$

- Since $z^{L+1} = \phi^{L+1}(W^L z^L)$, with $\phi^{L+1}(u) = \left(\phi^{L+1}(u_1), \cdots, \phi^{L+1}(u_{d_{out}})\right)^T$. We get $\frac{\partial z_j^{L+1}}{\partial z_i^L} = \phi'^{L+1}(W^L z^L)_i w_{ji}^L$ with $\phi'^{L+1}(u) = \left(\phi'^{L+1}(u_1), \cdots, \phi'^{L+1}(u_{d_{out}})\right)^T$.
- We easily deduce : $\frac{\partial e(f_\theta(x),y)}{\partial w_{ij}} = \sum_j \frac{\partial e(f_\theta(x),y)}{\partial z_j^{L+1}} \frac{\partial z_j^{L+1}}{\partial w_{ij}^L} = \sum_j \delta_j^{L+1} \frac{\partial z_j^{L+1}}{\partial w_{ij}^L}$

Matrix expression of the derivative computation

We can summarize the computation of the derivatives with matrices for an MLP with N layers :

- Propagation :
$$Z^{L+1} = \phi^{L+1}(W^L z^L), L = 1, \cdots, N.$$
- Back-propagation, with for two vectors u, v of $\mathbb{R}^d$, $u \times v = (u_1 v_1, \cdots, u_d v_d)^T$:
$$\delta^L = \left({W^L}^T \delta^{L+1} \right) \times {\phi'}^L(W^{L-1} z^{L-1}), L = N, \cdots, 2$$
$$\text{et } \delta^{N+1} := \left(\frac{\partial e(f_\theta(x), y)}{\partial z_i^{N+1}} \right)_{1 \le i \le d_{output}} \times {\phi'}^{L+1}(W^L z^L).$$
- Computation of the gradient for $L = 1, \cdots, N$:
$$\left(\frac{\partial e(f_\theta(x), y)}{\partial w_{ij}^L} \right)_{1 \le i \le d_{out}, 1 \le j \le d_{in}} = \delta^{L+1} \otimes Z^L,$$
$$\left(\frac{\partial e(f_\theta(x), y)}{\partial w_{i0}^L} \right)_{1 \le i \le d_{out}} = \delta^{L+1}.$$

  where $\otimes$ is the external product for two vectors : $v \in \mathbb{R}^n$, $u \in \mathbb{R}^m$,
$$v \otimes u = \begin{pmatrix} v_1 u_1 & \cdots & v_1 u_m \\ \vdots & \ldots & \vdots \\ v_n u_1 & \cdots & v_n u_m \end{pmatrix}$$

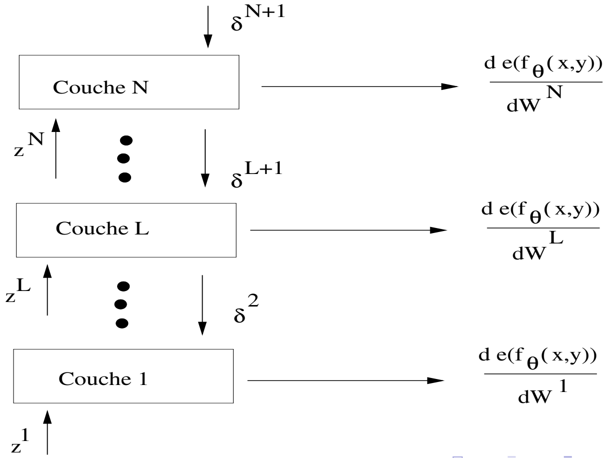
Application to elementary MLP

In our basic example, $N = 2$ and if we code y on the simplex : $0 := \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $1 := \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ we get :

- $z^1 = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$.
- $z^2 = \tanh(W^1 z^1)$.
- $z^3 = softmax(W^2 z^2)$.
- $\delta^3 = (z^3 - y) \times z^3 \times (1 - z^3)$.
- $\delta^2 = \left({W^2}^T \delta^3 \right) \times {\phi'}^2 (W^1 z^1) = \left({W^2}^T \delta^3 \right) \times \left(\begin{pmatrix} 1 \\ 1 \end{pmatrix} - z^2 \right) \times \left(\begin{pmatrix} 1 \\ 1 \end{pmatrix} + z^2 \right)$.
- $\left(\frac{\partial e(f_\theta(x),y)}{\partial w^2_{ij}} \right)_{1 \leq i \leq 2,\, 1 \leq j \leq 2} = \delta^3 \otimes z^2$, $\left(\frac{\partial e(f_\theta(x),y)}{\partial w^2_{i0}} \right)_{1 \leq i \leq 2} = \delta^3$
- $\left(\frac{\partial e(f_\theta(x),y)}{\partial w^1_{ij}} \right)_{1 \leq i \leq 2,\, 1 \leq j \leq 2} = \delta^2 \otimes z^1$, $\left(\frac{\partial e(f_\theta(x),y)}{\partial w^1_{i0}} \right)_{1 \leq i \leq 2} = \delta^2$.

Modular approach of derivatives computation

We see that, to compute the derivative of the cost function with respect to the weight, it is sufficient to propagate the z^L to backpropagate the δ^L and perform the computations inside the layer. We can thus build a complex network by stacking the layers and sending the results of the calculations of a layer to the possible following and previous layers.


δ^{N+1}
Couche N
$\frac{d\, e(f_\theta(x,y))}{dW^N}$
z^N
δ^{L+1}
Couche L
$\frac{d\, e(f_\theta(x,y))}{dW^L}$
z^L
δ^2
Couche 1
$\frac{d\, e(f_\theta(x,y))}{dW^1}$
z^1

Differential optimization

For each pair (x_t, y_t), we can compute, thanks to the backpropagation algorithm, the derivatives $\left(\frac{\partial e_t}{\partial \theta_i}\right)$, thus the gradient vector : ∇e_t. We can use this gradient to estimate the optimal weights in several ways In all cases, we randomly choose $\theta(0)$ (the components of $\theta(0)$ must be non-zero).

1. By a stochastic gradient descent (on-line).
2. By a batch gradient descent (off-line). This method is no longer used in Deep Learning because the number of parameters and the number of observations is too large.
3. By a mini-batch gradient descent of size B (between on-line and off-line). This is the preferred method for Deep Learning because it allows to parallelize the computations.

Differential optimization

For each pair (x_t, y_t), we can compute, thanks to the backpropagation algorithm, the derivatives $\left(\frac{\partial e_t}{\partial \theta_i}\right)$, thus the gradient vector : ∇e_t. We can use this gradient to estimate the optimal weights in several ways In all cases, we randomly choose $\theta(0)$ (the components of $\theta(0)$ must be non-zero).

1. By a stochastic gradient descent (on-line).
   - $\forall t \in \{1, \cdots, n\}$ update the parameter vector :

$$\theta(t+1) = \theta(t) - \varepsilon_{t+1} \nabla e_{t+1}$$

2. By a batch gradient descent (off-line). This method is no longer used in Deep Learning because the number of parameters and the number of observations is too large.
   - $\forall l \in \{1, \cdots, L\}$ update the parameter vector :

$$\theta(l+1) = \theta(l) - \varepsilon_{l+1} \frac{1}{n} \sum_{t=1}^{n} \nabla e_t$$

3. By a mini-batch gradient descent of size B (between on-line and off-line). This is the preferred method for Deep Learning because it allows to parallelize the computations.
   - $\forall l \in \{1, \cdots, L\}$ update the parameter vector :

$$\theta(l+1) = \theta(l) - \varepsilon_{l+1} \frac{1}{B} \sum_{t=l\times B+1}^{t=l\times(B+1)} \nabla e_t$$

Stochastic algorithm

$\theta(l+1) = \theta(l) - \varepsilon_l \frac{1}{B} \sum_{t=l\times B+1}^{t=l\times(B+1)} \nabla e_t$

This algorithm is the standard Deep Learning algorithm.

- If the sequence ε_t checks :

$$\sum_{t=1}^{\infty} \varepsilon_t = \infty \text{ et } \sum_{t=1}^{\infty} \varepsilon_t^2 < \infty$$

  and $(\theta(l))_{l\in\mathbb{N}}$ stays bounded.
- The stochastic gradient algorithm converges almost surely to a local minimum of $\theta \longmapsto C(\theta) = \mathbb{E}\left[e(f_\theta(X), Y)\right]$ when the number of observations increases towards infinity.
- In practice, ε_t remains constant over tens of iterations and decreases slowly while remaining greater than 10^{-6}.
- As we do not have an infinite number of observations, the algorithm has to go over the same data several times.

Acceleration of the stochastic gradient

- Momentum : This method stores the update at each step and computes the next one as a convex combination of the gradient for the new mini-batch and the previous modification
$$\begin{cases} \Delta(l+1) = \alpha\Delta(l) + (1-\alpha)\frac{1}{B}\sum_{t=l\times B+1}^{t=l\times(B+1)} \nabla e_t \\ \theta(l+1) = \theta(l) - \varepsilon_{l+1}\Delta(l+1) \end{cases}$$
- Adagrad or Adam : These algorithms improve the stochastic gradient method by automatically determining a learning rate for each parameter.
  - Let us write $G = \sum_1^k \nabla e_t(\nabla e_t)^T$ the estimated covariance matrix of ∇e_t.
  - The diagonal of G will be $G_{j,j} = \sum_{t=1}^k \nabla e_{t,j}^2$, which is updated after each mini-batch.
  - Let us write ε the step of the gradient algorithm. Each parameter will be updated according :
$$\theta_j = \theta_j - \frac{\varepsilon}{\sqrt{G_{j,j}}}(\nabla e_t)_j$$
  - Large parameter updates are mitigated while small changes are made with a higher learning rate.
  - Adam is an adaptation of Adagrad with an exponential forgetting of the old gradients.

Batch algorithm

$\theta(t+1) = \theta(t) - \varepsilon_{t+1} \frac{1}{n} \sum_{t=1}^{n} \nabla e_t$

If ε_{t+1} is small enough this algorithm is sure to converge to a local minimum of

$$\theta \longmapsto \mathbb{E}\left[C(f_\theta(X), Z)\right].$$

Batch algorithm

$\theta(t+1) = \theta(t) - \varepsilon_{t+1} \frac{1}{n} \sum_{t=1}^{n} \nabla e_t$

If ε_{t+1} is small enough this algorithm is sure to converge to a local minimum of

$$\theta \longmapsto \mathbb{E}\left[C(f_\theta(X), Z)\right].$$

Acceleration
Modification of the descent direction by estimating the inverse of the Hessian. The two most popular algorithms are BFGS and Levenberg-Marquardt (LM).

- The LM is only applicable to quadratic cost functions. It performs better than BFGS if the MLP has only one output.
- In the multidimensional case the BFGS is generally faster.
- Both algorithms provide an estimate of the inverse of the Hessian matrix.

These methods are generally no longer suitable for Deep Learning because the number of parameters and the number of observations are very large.

The overfitting

- Stochastic optimization algorithms work well in practice. They often converge to a very good local minimum of the cost function.
- As these models have many parameters, there are often more parameters than observations, so we must be careful about overfitting.
- Overfitting corresponds to a too close or exact fit to a particular data set. The model learns "by heart" the training data and cannot generalize on new data.
- Formally :
  - The algorithm minimizes “very well” $\theta \mapsto C_n(\theta) = \frac{1}{n}\sum_{t=1}^{n} e(f_\theta(x_t), y_t)$
  - but the chosen $\hat{\theta}$ will give a big generalization error $\mathbb{E}(e(f_{\hat{\theta}}(X), Y))$.

An example of overfitting
White noise

We draw a sample $((x_t(1), x_t(2), y_t)_{1\leq t\leq 30}$, where $(X_1, X_2, Y) \sim \mathcal{N}(0, I_3)$ and we try to predict the variable Y according to the couple (X_1, X_2). Since Y is independent of (X_1, X_2) and centered, the best prediction is zero : $\mathbb{E}\left(Y \,|X_1, X_2\right) = 0$.

```
library(nnet)
library(rgl)
set.seed(1)
x <- matrix(rnorm(60),30,2)
y <- matrix(rnorm(30),30,1)
res.mod <- nnet(x,y,size=10,nbstart=10,maxit=1000,linout=T)
xp <- runif(10000,min=min(x[,1]),max=max(x[,1]))
yp <- runif(10000,min=min(x[,2]),max=max(x[,2]))
matp <- cbind(xp,yp)
mod.pred <- predict(res.mod,matp)
plot3d(c(-2,-2,2,2),c(-2,2,-2,2),c(20,20,-20,-20),type="n",ax
points3d(xp,yp,mod.pred,col="green")
spheres3d(x[,1],x[,2],y,radius=0.5,col="red")
```

Another example of overfitting (1)

Still a white noise

- Now, we draw an i.i.d. sample of large dimension and size : $((X_t, Y_t)_{1\leq t\leq 100000}$, où $(X_t) \sim \mathcal{N}(0, I_{2000})$ and $Y_t \sim \mathcal{N}(0,1)$, is independant of X_t. We try to predict the variable Y according to the variable X. Since Y is independent of X and centered, the best prediction is zero : $\mathbb{E}\left(Y\,|X\right) = 0$.
- We train five different neural networks. All networks have two hidden layers and ReLU activation functions : $\phi(x) = \max(0,x)$. These models differ in the number of hidden units on each layer : from 2^3 to 2^7.
- Each model is optimized by the stochastic gradient method, with a mini-batch of size 64, a momentum of 0.9 and a constant step of 0.01.
- To evaluate the overfitting, we evaluate the estimated model on 100000 test data independent of the training data.

Another example of overfitting (2)
Still a white noise

The results are as follows :

TABLE – Comparison of the different architectures

Nb hidden units	Nb de parameters	data set	mean square error
2^3	16089	learning test	0.72 1.31
2^4	32305	learning test	0.47 1.67
2^5	65121	learning test	0.15 2.62
2^6	132289	learning test	0.06 2.16
2^7	272769	learning test	0.02 1.61

Regulation of the MLP

- We need a way to control the modeling power of the MLP.
- We limit its complexity, but we must be careful to let it be able to model a possibly very complicated problem.
- A compromise is sought between complexity (variance) and modeling (bias).

During deep network learning

- Weights decay
- Drop out
- Batch normalization
- Early stopping
- Mixup

Selection of the best model

- Best error on a validation set (hold out).
- This method is almost optimal if enough data are available for the classification.