Neural Networks and Introduction to Deep Learning

1 Introduction

Deep learning is a set of learning methods attempting to model data with complex architectures combining different non-linear transformations. The elementary bricks of deep learning are the neural networks, that are combined to form the deep neural networks.

These techniques have enabled significant progress in the fields of sound and image processing, including facial recognition, speech recognition, computer vision, automated language processing, text classification (for example spam recognition). Potential applications are very numerous. A spectacularly example is the AlphaGo program, which learned to play the go game by the deep learning method, and beated the world champion in 2016.

There exist several types of architectures for neural networks :

- The multilayer perceptrons, that are the oldest and simplest ones
- The Convolutional Neural Networks (CNN), particularly adapted for image processing
- The recurrent neural networks, used for sequential data such as text or times series.

They are based on deep cascade of layers. They need clever stochastic optimization algorithms, and initialization, and also a clever choice of the structure. They lead to very impressive results, although very few theoretical fondations are available till now.

The main references for this course are :

- Ian Goodfellow, Yoshua Bengio and Aaron Courville : http://www.deeplearningbook.org/
- Bishop (1995) : Neural networks for pattern recognition, Oxford University Press.
- The elements of Statistical Learning by T. Hastie et al [3].
- Hugo Larochelle (Sherbrooke): http ://www.dmi.usherb.ca/ larocheh/
- Christopher Olah's blog : http://colah.github.io/posts/2015-08-Understanding-LSTMs/
- Deep learning course, Charles Ollion et Olivier Grisel : https://github.com/m2dsupsdlclass/lectures-labs

2 Neural networks

An artificial neural network is an application, non linear with respect to its parameters θ that associates to an entry x an output $y = f(x, \theta)$. For the sake of simplicity, we assume that y is unidimensional, but it could also be multidimensional. This application f has a particular form that we will precise. The neural networks can be use for regression or classification. As usual in statistical learning, the parameters θ are estimated from a learning sample. The function to minimize is not convex, leading to local minimizers. The success of the method came from a universal approximation theorem due to Cybenko (1989) and Hornik (1991). Moreover, Le Cun (1986) proposed an efficient way to compute the gradient of a neural network, called backpropagation of the gradient, that allows to obtain a local minimizer of the quadratic criterion easily.

2.1 Artificial Neuron

An artificial neuron is a function f_j of the input $x = (x_1, \ldots, x_d)$ weighted by a vector of connection weights $w_j = (w_{j,1}, \ldots, w_{j,d})$, completed by a neuron bias b_j, and associated to an activation function ϕ, namely

$$y_j = f_j(x) = \phi(\langle w_j, x \rangle + b_j).$$

Several activation functions can be considered.

- The identity function

$$\phi(x) = x.$$

- The sigmoid function (or logistic)

$$\phi(x) = \frac{1}{1+\exp(-x)}.$$

- The hyperbolic tangent function ("tanh")

$$\phi(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)} = \frac{\exp(2x)-1}{\exp(2x)+1}.$$

- The hard threshold function

$$\phi_\beta(x) = \mathbf{1}_{x\geq\beta}.$$

- The Rectified Linear Unit (ReLU) activation function

$$\phi(x) = \max(0, x).$$

Here is a schematic representation of an artificial neuron where $\Sigma = \langle w_j, x\rangle + b_j$.

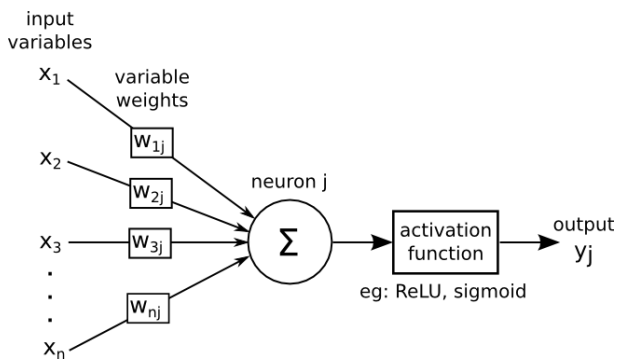


Figure 1: source: andrewjames turner.co.uk

The Figure 2 represents the activation function described above.

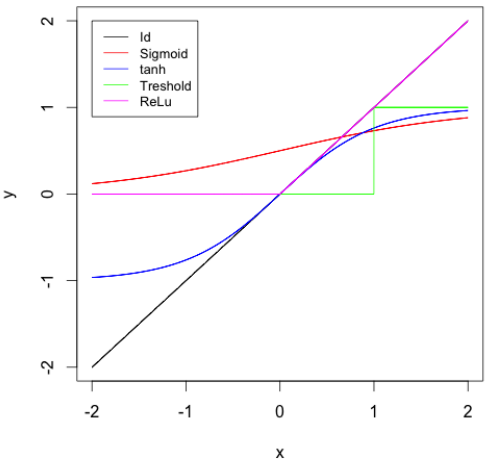


Figure 2: Activation functions

Historically, the sigmoid was the mostly used activation function since it is differentiable and allows to keep values in the interval $[0, 1]$. Nevertheless, it is problematic since its gradient is very close to 0 when $|x|$ is not close to 0. The Figure 3 represents the Sigmoid function and its derivative.

With neural networks with a high number of layers (which is the case for deep learning), this causes troubles for the backpropagation algorithm to estimate the parameter (backpropagation is explained in the following). This is why the sigmoid function was supplanted by the rectified linear function. This function is not differentiable in 0 but in practice this is not really a problem since the probability to have an entry equal to 0 is generally null. The ReLU function also has a sparsification effect. The ReLU function and its derivative are equal to 0 for negative values, and no information can be obtain in this case for such a

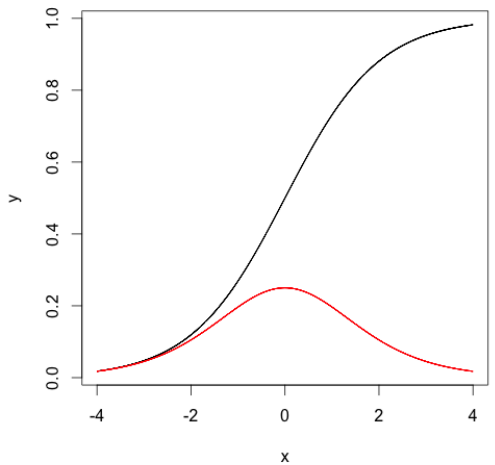


Figure 3: Sigmoid function (in black) and its derivatives (in red)

unit, this is why it is advised to add a small positive bias to ensure that each unit is active. Several variations of the ReLU function are considered to make sure that all units have a non vanishing gradient and that for $x < 0$ the derivative is not equal to 0. Namely

$$\phi(x) = \max(x, 0) + \alpha \min(x, 0)$$

where α is either a fixed parameter set to a small positive value, or a parameter to estimate.

2.2 Multilayer perceptron

A multilayer perceptron (or neural network) is a structure composed by several hidden layers of neurons where the output of a neuron of a layer becomes the input of a neuron of the next layer. Moreover, the output of a neuron can also be the input of a neuron of the same layer or of neuron of previous layers (this is the case for recurrent neural networks). On last layer, called output layer, we may apply a different activation function as for the hidden layers depending on the type of problems we have at hand : regression or classification. The Figure 4 represents a neural network with three input variables, one output variable, and two hidden layers.

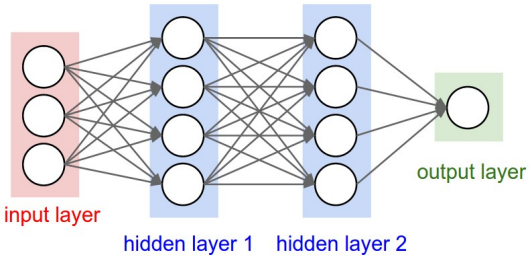


Figure 4: A basic neural network. Source : http://blog.christianperone.com

Multilayers perceptrons have a basic architecture since each unit (or neuron) of a layer is linked to all the units of the next layer but has no link with the neurons of the same layer. The parameters of the architecture are the number of hidden layers and of neurons in each layer. The activation functions are also to choose by the user. For the output layer, as mentioned previously, the activation function is generally different from the one used on the hidden layers. In the case of regression, we apply no activation function on the output layer. For binary classification, the output gives a prediction of $\mathbb{P}(Y = 1/X)$ since this value is in $[0, 1]$, the sigmoid activation function is generally considered. For multi-class classification, the output layer contains one neuron per class i, giving a prediction of $\mathbb{P}(Y = i/X)$. The sum of all these values has to be equal to 1. The multidimensional function *softmax* is generally used

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}.$$

Let us summarize the mathematical formulation of a multilayer perceptron with L hidden layers.
We set $h^{(0)}(x) = x$.

For $k = 1, \dots, L$ (hidden layers),

$$\begin{aligned} a^{(k)}(x) &= b^{(k)} + W^{(k)} h^{(k-1)}(x) \\ h^{(k)}(x) &= \phi(a^{(k)}(x)) \end{aligned}$$

For $k = L + 1$ (output layer),

$$\begin{aligned} a^{(L+1)}(x) &= b^{(L+1)} + W^{(L+1)} h^{(L)}(x) \\ h^{(L+1)}(x) &= \psi(a^{(L+1)}(x)) := f(x, \theta). \end{aligned}$$

where ϕ is the activation function and ψ is the output layer activation function (for example softmax for multiclass classification). At each step, $W^{(k)}$ is a matrix with number of rows the number of neurons in the layer k and number of columns the number of neurons in the layer $k - 1$.

2.3 Universal approximation theorem

Hornik (1991) showed that any bounded and regular function $\mathbb{R}^d \to \mathbb{R}$ can be approximated at any given precision by a neural network with one hidden layer containing a finite number of neurons, having the same activation function, and one linear output neuron. This result was earlier proved by Cybenko (1989) in the particular case of the sigmoid activation function. More precisely, Hornik's theorem can be stated as follows.

THEOREM 1. — *Let ϕ be a bounded, continuous and non decreasing (activation) function. Let K_d be some compact set in $\mathbb{R}^d$ and $\mathcal{C}(K_d)$ the set of continuous functions on K_d. Let $f \in \mathcal{C}(K_d)$. Then for all $\varepsilon > 0$, there exists $N \in \mathbb{N}$, real numbers v_i, b_i and $\mathbb{R}^d$-vectors w_i such that, if we define*

$$F(x) = \sum_{i=1}^{N} v_i \phi(\langle w_i, x \rangle + b_i)$$

then we have

$$\forall x \in K_d, |F(x) - f(x)| \leq \varepsilon.$$

This theorem is interesting from a theoretical point of view. From a practical point of view, this is not really useful since the number of neurons in the hidden layer may be very large. The strength of deep learning lies in the deep (number of hidden layers) of the networks.

2.4 Estimation of the parameters

Once the architecture of the network has been chosen, the parameters (the weights w_j and biases b_j) have to be estimated from a learning sample. As usual, the estimation is obtained by minimizing a loss function with a gradient descent algorithm. We first have to choose the loss function.

2.4.1 Loss functions

It is classical to estimate the parameters by maximizing the likelihood (or equivalently the logarithm of the likelihood). This corresponds to the minimization of the loss function which is the opposite of the log likelihood. Denoting θ the vector of parameters to estimate, we consider the expected loss function

$$L(\theta) = -\mathbb{E}_{(X,Y)\sim P}(\log(p_\theta(Y/X)).$$

If the model is Gaussian, namely if $p_\theta(Y/X = x) \sim \mathcal{N}(f(x, \theta), I)$, maximizing the likelihood is equivalent to minimize the quadratic loss

$$L(\theta) = \mathbb{E}_{(X,Y)\sim P}(\|Y - f(X, \theta)\|^2).$$

For binary classification, with $Y \in \{0, 1\}$, maximizing the log likelihood corresponds to the minimization of the cross-entropy. Setting $f(X, \theta)) = p_\theta(Y = 1/X)$,

$$L(\theta) = -\mathbb{E}_{(X,Y)\sim P}[Y \log(f(X, \theta)) + (1 - Y) \log(1 - f(X, \theta))].$$

This loss function is well adapted with the sigmoid activation function since the use of the logarithm avoids to have too small values for the gradient.
Finally, for a multi-class classification problem, we consider a generalization of the previous loss function to k classes

$$L(\theta) = -\mathbb{E}_{(X,Y)\sim P}[\sum_{j=1}^{k} \mathbf{1}_{Y=j} \log p_\theta(Y = j/X)].$$

Ideally we would like to minimize the classification error, but it is not smooth, this is why we consider the cross-entropy (or eventually a convex surrogate).

2.4.2 Penalized empirical risk

The expected loss can be written as

$$L(\theta) = \mathbb{E}_{(X,Y)\sim P}[\ell(f(X,\theta),Y)]$$

and it is associated to a loss function ℓ.
In order to estimate the parameters θ, we use a training sample $(X_i,Y_i)_{1\leq i\leq n}$ and we minimize the empirical loss

$$\tilde{L}_n(\theta) = \frac{1}{n}\sum_{i=1}^{n}\ell(f(X_i,\theta),Y_i)$$

eventually we add a regularization term. This leads to minimize the penalized empirical risk

$$L_n(\theta) = \frac{1}{n}\sum_{i=1}^{n}\ell(f(X_i,\theta),Y_i) + \lambda\Omega(\theta).$$

We can consider $\mathbb{L}^2$ regularization. Using the same notations as in Section 2.2,

$$\begin{aligned}\Omega(\theta) &= \sum_k\sum_i\sum_j (W_{i,j}^{(k)})^2\\ &= \sum_k \|W^{(k)}\|_F^2\end{aligned}$$

where $\|W\|_F$ denotes the Frobenius norm of the matrix W. Note that only the weights are penalized, the biases are not penalized. It is easy to compute the gradient of $\Omega(\theta)$:

$$\bigtriangledown_{W^{(k)}}\Omega(\theta) = 2W^{(k)}.$$

One can also consider $\mathbb{L}^1$ regularization, leading to parcimonious solutions :

$$\Omega(\theta) = \sum_k\sum_i\sum_j |W_{i,j}^{(k)}|.$$

In order to minimize the criterion $L_n(\theta)$, a **stochastic gradient descent algorithm** is used. In order to compute the gradient, a clever method, called **Backpropagation algorithm** is considered. It has been introduced by Rumelhart et al. (1988), it is still crucial for deep learning.

The stochastic gradient descent algorithm performs at follows :

- Initialization of $\theta = (W^{(1)}, b^{(1)}, \ldots, W^{(L+1)}, b^{(L+1)})$.
- For N iterations :
  - For each training data (X_i, Y_i),

$$\theta = \theta - \varepsilon\frac{1}{m}\sum_{i\in B}[\bigtriangledown_\theta \ell(f(X_i,\theta),Y_i) + \lambda\bigtriangledown_\theta\Omega(\theta)].$$

Note that, in the previous algorithm, we do not compute the gradient for the loss function at each step of the algorithm but only on a subset B of cardinality m (called a *batch*). This is what is classically done for big data sets (and for deep learning) or for sequential data. B is taken at random without replacement. An iteration over all the training examples is called an **epoch**. The numbers of epochs to consider is a parameter of the deep learning algorithms. The total number of iterations equals the number of epochs times the sample size n divided by m, the size of a batch. This procedure is called *batch learning*, sometimes, one also takes batches of size 1, reduced to a single training example (X_i, Y_i).

2.4.3 Backpropagation algorithm for regression with the quadratic loss

We consider the regression case and explain in this section how to compute the gradient of the empirical quadratic loss by the Backpropagation algorithm. To simplify, we do not consider here the penalization term, that can easily be added. Assuming that the output of the multilayer perceptron is of size K, and using the notations of Section 2.2, the empirical quadratic loss is proportional to

$$\sum_{i=1}^{n} R_i(\theta)$$

with

$$R_i(\theta) = \sum_{k=1}^{K}(Y_{i,k} - f_k(X_i,\theta))^2.$$

In a regression model, the output activation function ψ is generally the identity function, to be more general, we assume that

$$\psi(a_1, \ldots, a_K) = (g_1(a_1), \ldots, g_K(a_K))$$

where $g_1, \ldots, g_K$ are functions from $\mathbb{R}$ to $\mathbb{R}$. Let us compute the partial derivatives of R_i with respect to the weights of the output layer. Recalling that

$$a^{(L+1)}(x) = b^{(L+1)} + W^{(L+1)} h^{(L)}(x),$$

we get

$$\frac{\partial R_i}{\partial W_{k,m}^{(L+1)}} = -2(Y_{i,k} - f_k(X_i, \theta)) g_k'(a_k^{(L+1)}(X_i)) h_m^{(L)}(X_i).$$

Differentiating now with respect to the weights of the previous layer

$$\frac{\partial R_i}{\partial W_{m,l}^{(L)}} = -2 \sum_{k=1}^{K} (Y_{i,k} - f_k(X_i, \theta)) g_k'(a_k^{(L+1)}(X_i)) \frac{\partial a_k^{(L+1)}(X_i)}{\partial W_{m,l}^{(L)}}.$$

with

$$\begin{aligned} a_k^{(L+1)}(x) &= \sum_j W_{k,j}^{(L+1)} h_j^{(L)}(x), \\ h_j^{(L)}(x) &= \phi\left(b_j^{(L)} + \langle W_j^{(L)}, h^{(L-1)}(x) \rangle\right). \end{aligned}$$

This leads to

$$\frac{\partial a_k^{(L+1)}(x)}{\partial W_{m,l}^{(L)}} = W_{k,m}^{(L+1)} \phi'\left(b_m^{(L)} + \langle W_m^{(L)}, h^{(L-1)}(x) \rangle\right) h_l^{(L-1)}(x).$$

Let us introduce the notations

$$\begin{aligned} \delta_{k,i} &= -2(Y_{i,k} - f_k(X_i, \theta)) g_k'(a_k^{(L+1)}(X_i)) \\ s_{m,i} &= \phi'\left(a_m^{(L)}(X_i)\right) \sum_{k=1}^{K} W_{k,m}^{(L+1)} \delta_{k,i}. \end{aligned}$$

Then we have

$$\frac{\partial R_i}{\partial W_{k,m}^{(L+1)}} = \delta_{k,i} h_m^{(L)}(X_i) \quad (1)$$

$$\frac{\partial R_i}{\partial W_{m,l}^{(L)}} = s_{m,i} h_l^{(L-1)}(X_i), \quad (2)$$

known as the *backpropagation equations*. The values of the gradient are used to update the parameters in the gradient descent algorithm. At step $r+1$, we have :

$$\begin{aligned} W_{k,m}^{(L+1,r+1)} &= W_{k,m}^{(L+1,r)} - \varepsilon_r \sum_{i \in B} \frac{\partial R_i}{\partial W_{k,m}^{(L+1,r)}} \\ W_{m,l}^{(L,r+1)} &= W_{m,l}^{(L,r)} - \varepsilon_r \sum_{i \in B} \frac{\partial R_i}{\partial W_{m,l}^{(L,r)}} \end{aligned}$$

where B is a batch (either the n training sample or a subsample, eventually of size 1) and $\varepsilon_r > 0$ is the learning rate that satisfies $\varepsilon_r \to 0$, $\sum_r \varepsilon_r = \infty$, $\sum_r \varepsilon_r^2 < \infty$, for example $\varepsilon_r = 1/r$.

We use the Backpropagation equations to compute the gradient by a two pass algorithm. In the *forward pass*, we fix the value of the current weights $\theta^{(r)} = (W^{(1,r)}, b^{(1,r)}, \ldots, W^{(L+1,r)}, b^{(L+1,r)})$, and we compute the predicted values $f(X_i, \theta^{(r)})$ and all the intermediate values $(a^{(k)}(X_i), h^{(k)}(X_i) = \phi(a^{(k)}(X_i)))_{1 \leq k \leq L+1}$ that are stored.

Using these values, we compute during the *backward pass* the quantities $\delta_{k,i}$ and $s_{m,i}$ and the partial derivatives given in Equations 1 and 2. We have computed the partial derivatives of R_i only with respect to the weights of the output layer and the previous ones, but we can go on to compute the partial derivatives of R_i with respect to the weights of the previous hidden layers. In the back propagation algorithm, each hidden layer gives and receives informations from the neurons it is connected with. Hence, the algorithm is adapted for parallel computations. The computations of the partial derivatives involve the function ϕ', where ϕ is the activation functions. ϕ' can generally be expressed in a simple way for classical activations functions. Indeed for the sigmoid function

$$\phi(x) = \frac{1}{1 + \exp(-x)}, \quad \phi'(x) = \phi(x)(1 - \phi(x)).$$

For the hyperbolic tangent function ("tanh")

$$\phi(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}, \quad \phi'(x) = 1 - \phi^2(x).$$

The backpropagation algorithm is also used for classification with the cross entropy as explained in the next section.

2.4.4 Backpropagation algorithm for classification with the cross entropy

We consider here a K class classification problem. The output of the MLP is $f(x) = \begin{pmatrix} \mathbb{P}(Y = 1/x) \\ . \\ . \\ \mathbb{P}(Y = K/x) \end{pmatrix}$. We assume that the output activation function is the *softmax* function.

$$\text{softmax}(x_1, \ldots, x_K) = \frac{1}{\sum_{k=1}^K e^{x_i}}(e^{x_1}, \ldots, e^{x_K}).$$

Let us make some useful computations to compute the gradient.

$$\begin{aligned} \frac{\partial \text{softmax}(\boldsymbol{x})_i}{\partial x_j} &= \text{softmax}(\boldsymbol{x})_i(1 - \text{softmax}(\boldsymbol{x})_i) \text{ if } i = j \\ &= -\text{softmax}(\boldsymbol{x})_i \text{softmax}(\boldsymbol{x})_j \text{ if } i \neq j \end{aligned}$$

We introduce the notation

$$(f(x))_y = \sum_{k=1}^K \mathbf{1}_{y=k}(f(x))_k,$$

where $(f(x))_k$ is the kth component of $f(x)$: $(f(x))_k = \mathbb{P}(Y = k/x)$. Then we have

$$-\log(f(x))_y = -\sum_{k=1}^K \mathbf{1}_{y=k} \log(f(x))_k = \ell(f(x), y),$$

for the loss function ℓ associated to the cross-entropy.
Using the notations of Section 2.2, we want to compute the gradients

$$\text{Output weights} \frac{\partial \ell(f(x), y)}{\partial W_{i,j}^{(L+1)}} \quad \text{Output biases} \frac{\partial \ell(f(x), y)}{\partial b_i^{(L+1)}}$$

$$\text{Hidden weights} \frac{\partial \ell(f(x), y)}{\partial W_{i,j}^{(h)}} \quad \text{Hidden biases} \frac{\partial \ell(f(x), y)}{\partial b_i^{(h)}}$$

for $1 \leq h \leq L$. We use the chain-rule : if $z(x) = \phi(a_1(x), \ldots, a_J(x))$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial a_j} \frac{\partial a_j}{\partial x_i} = \langle \nabla \phi, \frac{\partial \boldsymbol{a}}{\partial x_i} \rangle.$$

Hence we have

$$\frac{\partial \ell(f(x), y)}{\partial (a^{(L+1)}(x))_i} = \sum_j \frac{\partial \ell(f(x), y)}{\partial f(x)_j} \frac{\partial f(x)_j}{\partial (a^{(L+1)}(x))_i}.$$

$$\frac{\partial \ell(f(x), y)}{\partial f(x)_j} = \frac{-\mathbf{1}_{y=j}}{(f(x))_y}.$$

$$\begin{aligned} \frac{\partial \ell(f(x), y)}{\partial (a^{(L+1)}(x))_i} &= -\sum_j \frac{\mathbf{1}_{y=j}}{(f(x))_y} \frac{\partial \text{softmax}(a^{(L+1)}(x))_j}{\partial (a^{(L+1)}(x))_i} \\ &= -\frac{1}{(f(x))_y} \frac{\partial \text{softmax}(a^{(L+1)}(x))_y}{\partial (a^{(L+1)}(x))_i} \\ &= -\frac{1}{(f(x))_y} \text{softmax}(a^{(L+1)}(x))_y (1 - \text{softmax}(a^{(L+1)}(x))_y) \mathbf{1} y = i \\ &+ \frac{1}{(f(x))_y} \text{softmax}(a^{(L+1)}(x))_i \text{softmax}(a^{(L+1)}(x))_y \mathbf{1}_{y \neq i} \end{aligned}$$

$$\frac{\partial \ell(f(x), y)}{\partial (a^{(L+1)}(x))_i} = (-1 + f(x)_y) \mathbf{1} y = i + f(x)_i \mathbf{1} y \neq i.$$

Hence we obtain

$$\nabla_{a^{(L+1)}(x)} \ell(f(x), y) = f(x) - e(y),$$

where, for $y \in \{1, 2, \ldots, K\}$, $e(y)$ is the $\mathbb{R}^K$ vector with i th component $\mathbf{1}_{i=y}$. We now obtain easily the partial derivative of the loss function with respect to the output bias. Since

$$\frac{\partial((a^{(L+1)}(x)))_j}{\partial(b^{(L+1)})_i} = \mathbf{1}_{i=j},$$

$$\nabla_{b^{(L+1)}} \ell(f(x), y) = f(x) - e(y), \tag{3}$$

Let us now compute the partial derivative of the loss function with respect to the output weights.

$$\frac{\partial \ell(f(x), y)}{\partial W_{i,j}^{(L+1)}} = \sum_k \frac{\partial \ell(f(x), y)}{\partial (a^{(L+1)}(x))_k} \frac{\partial (a^{(L+1)}(x))_k}{\partial W_{i,j}^{(L+1)}}$$

and

$$\frac{\partial (a^{(L+1)}(x))_k}{\partial W_{i,j}^{(L+1)}} = a^{(L)}(x))_j \mathbf{1}_{i=k}.$$

Hence

$$\nabla_{W^{(L+1)}} \ell(f(x), y) = (f(x) - e(y))(a^{(L)}(x))'. \tag{4}$$

Let us now compute the gradient of the loss function at hidden layers. We use the chain rule

$$\frac{\partial \ell(f(x), y)}{\partial (h^{(k)}(x))_j} = \sum_i \frac{\partial \ell(f(x), y)}{\partial (a^{(k+1)}(x))_i} \frac{\partial (a^{(k+1)}(x))_i}{\partial (h^{(k)}(x))_j}$$

We recall that

$$(a^{(k+1)}(x))_i = b_i^{(k+1)} + \sum_j W_{i,j}^{(k+1)} (h^{(k)}(x))_j.$$

Hence

$$\frac{\partial \ell(f(x), y)}{\partial h^{(k)}(x)_j} = \sum_i \frac{\partial \ell(f(x), y)}{\partial a^{(k+1)}(x)_i} W_{i,j}^{(k+1)}$$

$$\nabla_{h^{(k)}(x)} \ell(f(x), y) = (W^{(k+1)})' \nabla_{a^{(k+1)}(x)} \ell(f(x), y).$$

Recalling that $h^{(k)}(x)_j = \phi(a^{(k)}(x)_j)$,

$$\frac{\partial \ell(f(x), y)}{\partial a^{(k)}(x)_j} = \frac{\partial \ell(f(x), y)}{\partial h^{(k)}(x)_j} \phi'(a^{(k)}(x)_j).$$

Hence,

$$\nabla_{a^{(k)}(x)} \ell(f(x), y) = \nabla_{h^{(k)}(x)} \ell(f(x), y) \odot (\phi'(a^{(k)}(x)_1), \ldots, \phi'(a^{(k)}(x)_j), \ldots)'$$

where $\odot$ denotes the element-wise product. This leads to

$$\begin{aligned} \frac{\partial \ell(f(x), y)}{\partial W_{i,j}^{(k)}} &= \frac{\partial \ell(f(x), y)}{\partial a^{(k)}(x)_i} \frac{\partial a^{(k)}(x)_i}{\partial W_{i,j}^{(k)}} \\ &= \frac{\partial \ell(f(x), y)}{\partial a^{(k)}(x)_i} h_j^{(k-1)}(x) \end{aligned}$$

Finally, the gradient of the loss function with respect to hidden weights is

$$\nabla_{W^{(k)}} \ell(f(x), y) = \nabla_{a^{(k)}(x)} \ell(f(x), y) h^{(k-1)}(x)'. \tag{5}$$

The last step is to compute the gradient with respect to the hidden biases. We simply have

$$\frac{\partial \ell(f(x), y)}{\partial b_i^{(k)}} = \frac{\partial \ell(f(x), y)}{\partial a^{(k)}(x)_i}$$

and

$$\nabla_{b^{(k)}} \ell(f(x), y) = \nabla_{a^{(k)}(x)} \ell(f(x), y). \tag{6}$$

We can now summarize the backpropagation algorithm.

- **Forward pass**: we fix the value of the current weights $\theta^{(r)} = (W^{(1,r)}, b^{(1,r)}, \ldots, W^{(L+1,r)}, b^{(L+1,r)})$, and we compute the predicted values $f(X_i, \theta^{(r)})$ and all the intermediate values $(a^{(k)}(X_i), h^{(k)}(X_i) = \phi(a^{(k)}(X_i)))_{1 \leq k \leq L+1}$ that are stored.
- **Backpropagation algorithm**:
  - Compute the output gradient $\nabla_{a^{(L+1)}(x)} \ell(f(x), y) = f(x) - e(y)$.
  - For $k = L + 1$ to 1

* Compute the gradient at the hidden layer k

$$\begin{aligned}\nabla_{W^{(k)}}\ell(f(x),y) &= \nabla_{a^{(k)}(x)}\ell(f(x),y)h^{(k-1)}(x)'\\ \nabla_{b^{(k)}}\ell(f(x),y) &= \nabla_{a^{(k)}(x)}\ell(f(x),y)\end{aligned}$$

* Compute the gradient at the previous layer

$$\nabla_{h^{(k-1)}(x)}\ell(f(x),y) = (W^{(k)})'\,\nabla_{a^{(k)}(x)}\,\ell(f(x),y)$$

and

$$\begin{aligned}\nabla_{a^{(k-1)}(x)}\ell(f(x),y) &= \nabla_{h^{(k-1)}(x)}\ell(f(x),y)\\ &\quad\odot(\ldots,\phi'(a^{(k-1)}(x)_j),\ldots)'\end{aligned}$$

2.4.5 Initialization

The input data have to be normalized to have approximately the same range. The biases can be initialized to 0. The weights cannot be initialized to 0 since for the $\tanh$ activation function, the derivative at 0 is 0, this is a saddle point. They also cannot be initialized with the same values, otherwise, all the neurons of a hidden layer would have the same behaviour. We generally initialize the weights at random : the values $W_{i,j}^{(k)}$ are i.i.d. Uniform on $[-c,c]$ with possibly $c=\frac{\sqrt{6}}{N_k+N_{k-1}}$ where N_k is the size of the hidden layer k. We also sometimes initialize the weights with a normal distribution $\mathcal{N}(0,0.01)$ (see Gloriot and Bengio, 2010).

2.4.6 Optimization algorithms

Many algorithms can be used to minimize the loss function, all of them have hyperparameters, that have to be calibrated, and have an important impact on the convergence of the algorithms. The elementary tool of all these algorithms is the Stochastic Gradient Descent (SGD) algorithm. It is the most simple one:

$$\theta_i^{new}=\theta_i^{old}-\varepsilon\frac{\partial L}{\partial\theta_i}(\theta_i^{old}),$$

where ε is the *learning rate* , and its calibration is very important for the convergence of the algorithm. If it is too small, the convergence is very slow and the optimization can be blocked on a local minimum. If the learning rate is too large, the network will oscillate around an optimum without stabilizing and converging. A classical way to proceed is to adapt the learning rate during the training : it is recommended to begin with a "large " value of ϵ, (for example 0.1) and to reduce its value during the successive iterations. However, there is no general rule on how to adjust the learning rate, and this is more the experience of the engineer concerning the observation of the evolution of the loss function that will give indications on the way to proceed.
The stochasticity of the SGD algorithm lies in the computation of the gradient. Indeed, we consider *batch learning* : at each step, m training examples are randomly chosen without replacement and the mean of the m corresponding gradients is used to update the parameters. An *epoch* corresponds to a pass through all the learning data, for example if the batch size m is $1/100$ times the sample size n, an epoch corresponds to 100 batches. We iterate the process on a certain number nb of *epochs* that is fixed in advance. If the algorithm did not converge after nb epochs, we have to continue for nb' more epochs. Another stopping rule, called *early stopping* is also used : it consists in considering a validation sample, and stop learning when the loss function for this validation sample stops to decrease. *Batch learning* is used for computational reasons, indeed, as we have seen, the backpropagation algorithm needs to store all the intermediate values computed at the forward step, to compute the gradient during the backward pass, and for big data sets, such as millions of images, this is not feasible, all the more that the deep networks have millions of parameters to calibrate. The batch size m is also a parameter to calibrate. Small batches generally lead to better generalization properties. The particular case of batches of size 1 is called *On-line Gradient Descent*. The disadvantage of this procedure is the very long computation time. Let us summarize the classical SGD algorithm.

ALGORITHM 1 **Stochastic Gradient Descent algorithm**

- *Fix the parameters ε : learning rate, m : batch size, nb : number of epochs.*
- *For $l=1$ to nb epochs*
- *For $l=1$ to n/m,*

– *Take a random batch of size m without replacement in the learning sample :* $(X_i, Y_i)_{i\in B_l}$

– *Compute the gradients with the backpropagation algorithm*

$$\tilde{\nabla}_\theta = \frac{1}{m}\sum_{i\in B_l} \nabla_\theta \ell(f(X_i,\theta), Y_i).$$

– *Update the parameters*

$$\theta^{new} = \theta^{old} - \varepsilon\tilde{\nabla}_\theta.$$

Since the choice of the learning rate is delicate and very influent on the convergence of the SGD algorithm, variations of the algorithm have been proposed. They are less sensitive to the learning rate. The principle is to add a correction when we update the gradient, called **momentum**. The method is due to Polyak (1964) [9].

$$(\tilde{\nabla}_\theta)^{(r)} = \gamma(\tilde{\nabla}_\theta)^{(r-1)} + \frac{\varepsilon}{m}\sum_{i\in B_l} \nabla_\theta \ell(f(X_i,\theta^{(r-1)}), Y_i).$$

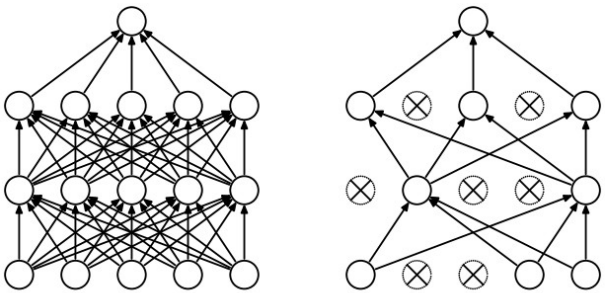
$$\theta^{(r)} = \theta^{(r-1)} - (\tilde{\nabla}_\theta)^{(r)}.$$

This method allows to attenuate the oscillations of the gradient.
In practice, a more recent version of the momentum due to Nesterov (1983) [8] and Sutskever et al. (2013) [11] is considered, it is called **Nesterov accelerated gradient** :

$$(\tilde{\nabla}_\theta)^{(r)} = \gamma(\tilde{\nabla}_\theta)^{(r-1)} + \frac{\varepsilon}{m}\sum_{i\in B_l} \nabla_\theta \ell(f(X_i,\theta^{(r-1)} - \gamma(\tilde{\nabla}_\theta)^{(r-1)}), Y_i).$$

$$\theta^{(r)} = \theta^{(r-1)} - (\tilde{\nabla}_\theta)^{(r)}.$$

There exist also more sophisticated algorithms, called **adaptive algorithms**. One of the most famous is the **RMSProp** algorithm, due to Hinton (2012) [2] or **Adam** (for Adaptive Moments) algorithm, see Kingma and Ba (2014) [5]. To conclude, let us say a few words about regularization. We have already mentioned $\mathbb{L}^2$ or $\mathbb{L}^1$ penalization; we have also mentioned early stopping. For deep learning, the mostly used method is the **dropout**. It was introduced by Hinton et al. (2012), [2]. With a certain probability p, and independently of the others, each unit of the network is set to 0. The probability p is another hyperparameter. It is classical to set it to 0.5 for units in the hidden layers, and to 0.2 for the entry layer. The computational cost is weak since we just have to set to 0 some weights with probability p. This method improves significantly the generalization properties of deep neural networks and is now the most popular regularization method in this context. The disadvantage is that training is much slower (it needs to increase the number of epochs). Ensembling models (aggregate several models) can also be used. It is also classical to use data augmentation or Adversarial examples.



(a) Standard Neural Net (b) After applying dropout.

Figure 5: Dropout - source: http://blog.christianperone.com/

3 Convolutional neural networks

For some types of data, especially for images, multilayer perceptrons are not well adapted. Indeed, they are defined for vectors as input data, hence, to apply them to images, we should transform the images into vectors, loosing by the way the spatial informations contained in the images, such as forms. Before the development of deep learning for computer vision, learning was based on the extraction of variables of interest, called *features*, but these methods need a lot of experience for image processing. The **convolutional neural networks (CNN)** introduced by LeCun [13] have revolutionized image processing, and

removed the manual extraction of features. CNN act directly on matrices, or even on tensors for images with three RGB color chanels. CNN are now widely used for image classification, image segmentation, object recognition, face recognition ..

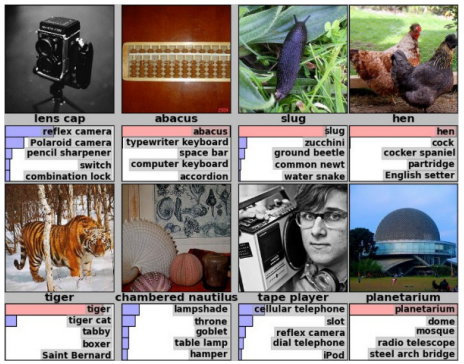


Figure 6: Image annotation. Source : http://danielnouri.org/media/deep-learning-whales-krizhevsky-lsvrc-2012-predictions.jpg

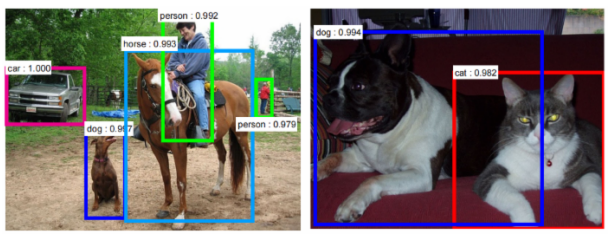


Figure 7: Image Segmentation. Source : http://static.open-open.com/lib/uploadImg/20160114/20160114205542-482.png

3.0.7 Layers in a CNN

A Convolutional Neural Network is composed by several kinds of layers, that are described in this section : convolutional layers, pooling layers and fully connected layers.

3.0.8 Convolution layer

The discrete convolution between two functions f and g is defined as

$$(f * g)(x) = \sum_t f(t)g(x+t).$$

For 2-dimensional signals such as images, we consider the 2D-convolutions

$$(K * I)(i,j) = \sum_{m,n} K(m,n)I(i+n, j+m).$$

K is a convolution kernel applied to a 2D signal (or image) I.

As shown in Figure 8, the principle of 2D convolution is to drag a convolution kernel on the image. At each position, we get the convolution between the kernel and the part of the image that is currently treated. Then, the kernel moves by a number s of pixels, s is called the *stride*. When the stride is small, we get redondant information. Sometimes, we also add a *zero padding*, which is a margin of size p containing zero values around the image in order to control the size of the output. Assume that we apply C_0 kernels (also called filters), each of size $k \times k$ on an image. If the size of the input image is $W_i \times H_i \times C_i$ (W_i denotes the width, H_i the height, and C_i the number of channels, typically $C_i = 3$), the volume of the output is $W_0 \times H_0 \times C_0$, where C_0 corresponds to the number of kernels that we consider, and

$$W_0 = \frac{W_i - k + 2p}{s} + 1$$

$$H_0 = \frac{H_i - k + 2p}{s} + 1.$$

If the image has 3 channels and if K_l ($l = 1, \ldots, C_0$) denote $5 \times 5 \times 3$ kernels (where 3 corresponds to the number of channels of the input image), the

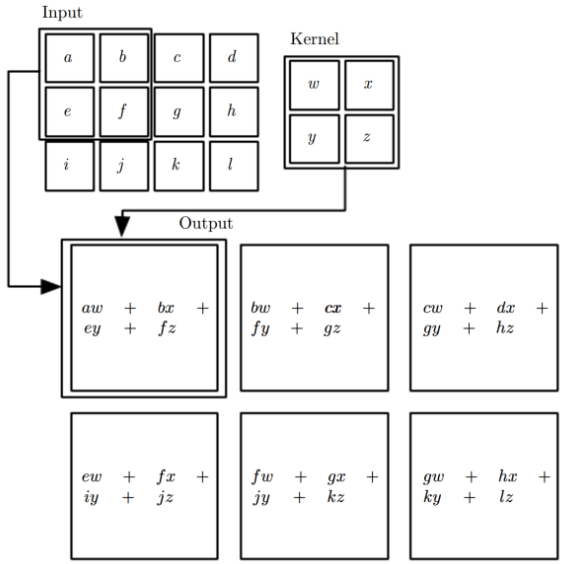


Figure 8: 2D convolution. Source :https://i2.wp.com/khshim.files.wordpress.com/2016/10/2d-convolution-example.png

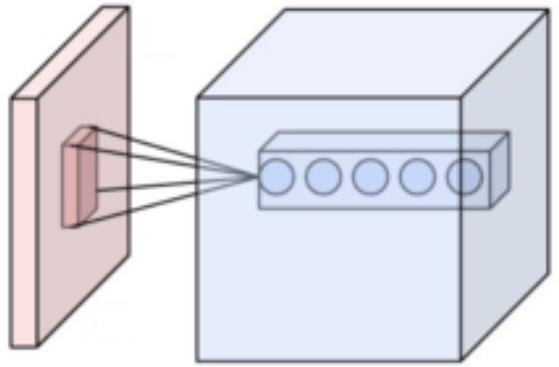


Figure 9: 2D convolution - Units corresponding to the same position but at various depths : each unit applies a different kernel on the same patch of the image. Source : https://upload.wikimedia.org/wikipedia/commons/thumb/6/68/Conv-layer.png/ 231px-Conv-layer.png

convolution with the image I with the kernel K_l corresponds to the formula:

$$K_l * I(i,j) = \sum_{c=0}^{2} \sum_{n=0}^{4} \sum_{m=0}^{4} K_l(n,m,c) I(i+n-2, i+m-2, c).$$

More generally, for images with C^i channels, the shape of the kernel is (k,k,C^i,C^0) where C^0 is the number of output channels (number of kernels) that we consider. This is $(5,5,3,2)$ in Figure 10. The number of parameter associated with a kernel of shape (k,k,C^i,C^0) is $(k \times k \times C^i + 1) \times C^0$.

The convolution operations are combined with an activation function ϕ (generally the Relu activation function) : if we consider a kernel K of size $k \times k$, if $\boldsymbol{x}$ is a $k \times k$ patch of the image, the activation is obtained by sliding the $k \times k$ window and computing $z(\boldsymbol{x}) = \phi(K * x + b)$, where b is a bias.

This is in the convolution layer that we find the strength of the CNN, indeed, the CNN will learn the filters (or kernels) that are the most useful for the task that we have to do (such as classification). Another advantage is that several convolution layers can be considered : the output of a convolution becomes the input of the next one.

3.0.9 Pooling layer

CNN also have *pooling* layers, which allow to reduce the dimension, also referred as *subsampling*, by taking the mean or the maximum on patches of the image (mean-pooling or max-pooling). Like the convolutional layers, pooling layers acts on small patches of the image, we also have a stride. If we consider 2×2 patches, over which we take the maximum value to define the output layer, and a stride $s = 2$, we divide by 2 the width and height of the image. Of course, it is also possible to reduce the dimension with the convolutional layer, by taking a stride larger than 1, and without zero padding

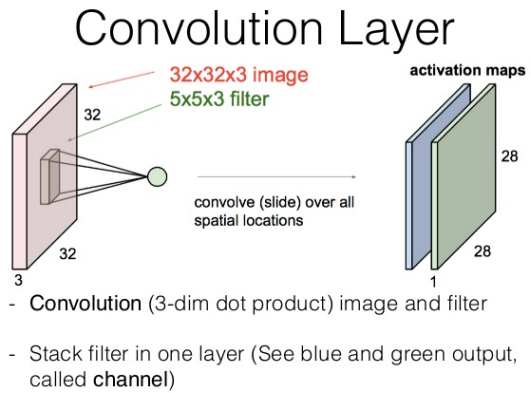


Figure 10: Source :http://image.slidesharecdn.com/

but another advantage of the pooling is that it makes the network less sensitive to small translations of the input images.

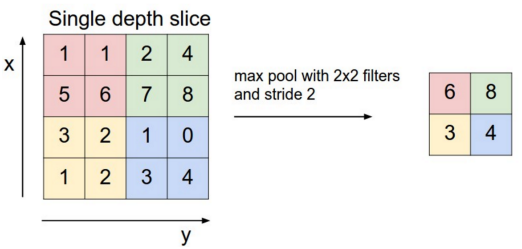


Figure 11: Maxpooling and effect on the dimension - Source : http://www.wildml.com/wp-content/uploads/2015/11/Screen-Shot-2015-11-05-at-2.18.38-PM.png

3.0.10 Fully connected layers

After several convolution and pooling layers, the CNN generally ends with several *fully connected* layers. The tensor that we have at the output of these layers is transformed into a vector and then we add several perceptron layers.

3.1 Architectures

We have described the different types of layers composing a CNN. We now present how this layers are combined to form the architecture of the network. Choosing an architecture is very complex and this is more engineering that an exact science. It is therefore important to study the architectures that have proved to be effective and to draw inspiration from these famous examples. In the most classical CNN, we chain several times a convolution layer followed by a pooling layer and we add at the end fully connected layers. The **LeNet** network, proposed by the inventor of the CNN, Yann LeCun [12] is of this type, as shown in Figure 12. This network was devoted to digit recognition. It is composed only on few layers and few filters, due to the computer limitations at that time.

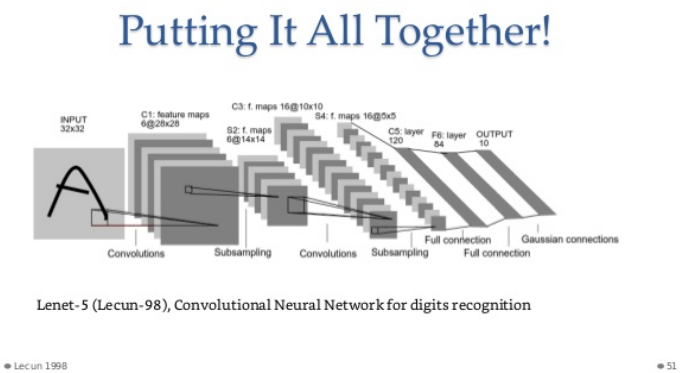


Figure 12: Architecture of the network Le Net. LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998)

A few years later, with the appearance of GPU (Graphical Processor Unit) cards, much more complex architectures for CNN have been proposed, like the network **AlexNet** (see [6]) that won the ImageNet competition and for which a simplified version is presented in Figure 13. This competition was devoted to the classification of one million of color images onto 1000 classes. The resolution of images was 224×224. AlexNet is composed of 5 convolution layers, 3 max-pooling 2×2 layers and fully connected layers. As showed if Figure 13, the kernel shape of the first convolution layer is $(11, 11, 3, 96)$ with a stride of $s = 4$, and the first output shape is $(55, 55, 96)$.

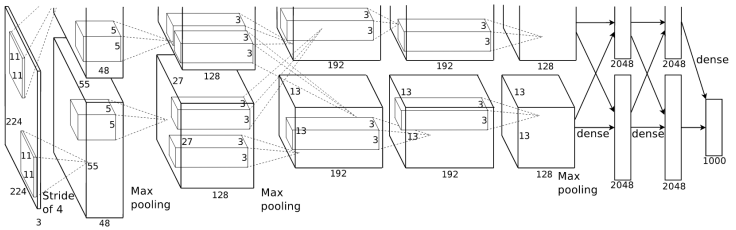


Figure 13: Architecture of the network AlexNet. Krizhevsky, A. et al (2012). Source : https://kratzert.github.io/

We detail in the following tabular the architecture of the network :

Input	227 * 227 * 3			
Conv 1	55*55*96	96	11 *11	filters at stride 4, pad 0
Max Pool 1	27*27*96		3 *3	filters at stride 2
Conv 2	27*27*256	256	5*5	filters at stride 1, pad 2
Max Pool 2	13*13*256		3 *3	filters at stride 2
Conv 3	13*13*384	384	3*3	filters at stride 1, pad 1
Conv 4	13*13*384	384	3*3	filters at stride 1, pad 1
Conv 5	13*13*256	256	3*3	filters at stride 1, pad 1
Max Pool 3	6*6*256		3 *3	filters at stride 2
FC1	4096	4096	neurons	
FC2	4096	4096	neurons	
FC3	1000	1000	neurons	(softmax logits)

Figure 14 presents another example.

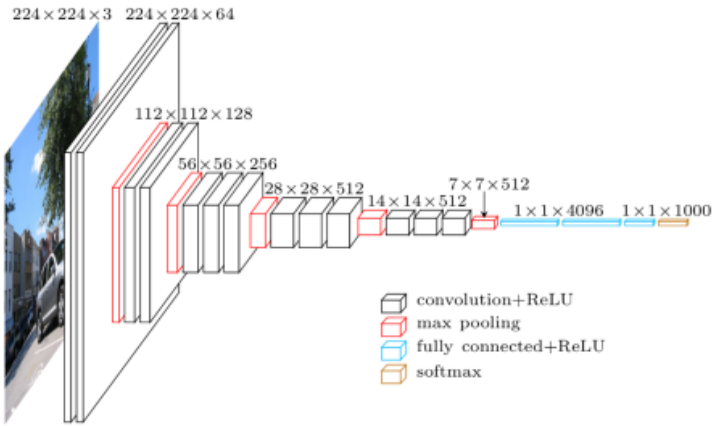


Figure 14: Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition (2014). Source : http://www.cs.toronto.edu/ frossard/post/vgg16/vgg16.png

The network that won the competition in 2014 is the network **GoogLeNet** [1], which is a new kind of CNN, not only composed on successive convolution and pooling layers, but also on new modules called *Inception*, which are some kind of network in the network. An example is represented in Figure 15.

The most recent innovations concern the **ResNet** networks (see [1]). The originality of the ResNets is to add a connection linking the input of a layer (or a set of layers) with its output. In order to reduce the number of parameters, the ResNets do not have fully connected layers. GoogleNet and ResNet are much deeper than the previous CNN, but contain much less parameters. They are nevertheless much costly in memory than more classical CNN such as VGG or AlexNet.

Figure 17 shows a comparison of the deepth and of the performances of the different networks, on the ImageNet challenge.

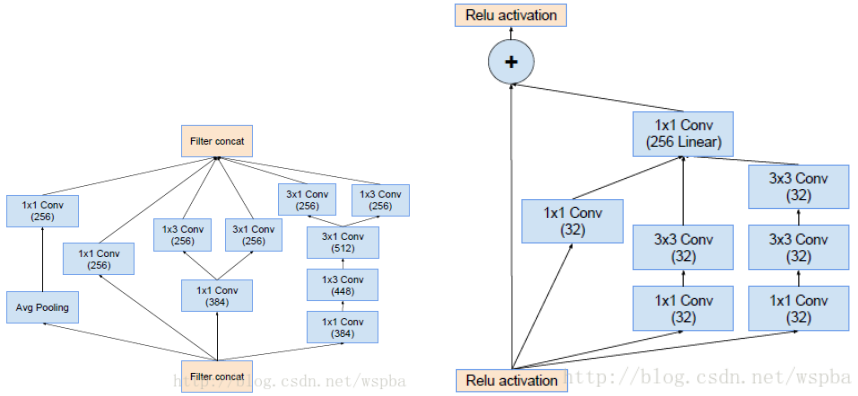


Figure 15: Inception modules, Szegedy et al. (2016)

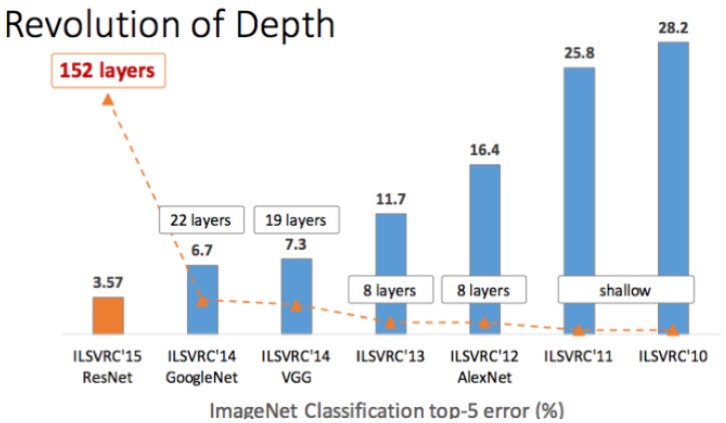


Figure 17: Evolution of the deepth of the CNN and the test error

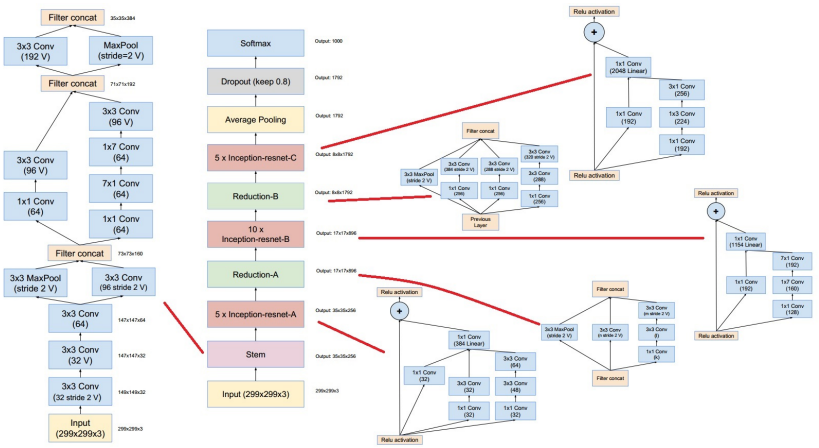


Figure 16: Inception-v4, Inception-resnet (Szegedy, C. et al. , 2016 [1])

were developed in the 1980's, a hidden layer at time t depends on the entry at time t, x_t but also on the same hidden layer at time $t-1$ or on the output at time $t-1$. We therefore have a loop from a hidden layer to itself or from the output to the hidden layer as shown in Figure 18. RNN may seem, at the first glance,

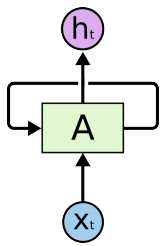


Figure 18: Diagram of a RNN. Source : Understanding LSTM Networks by Christopher Olah - http://colah.github.io/posts/2015-08-Understanding-LSTMs/

4 Recurrent neural networks

In order to infer sequential data such as text or time series, **Recurrent Neural Networks (RNN)** are considered. The most simple recurrent networks very different from classical neural network. In fact, this is not the case. RNN

can be seem as multiple copies of the same network (As in Figure 18 and 19), each passing information to its successor. This is the unrolled representation of RNN, shown in Figure 19.

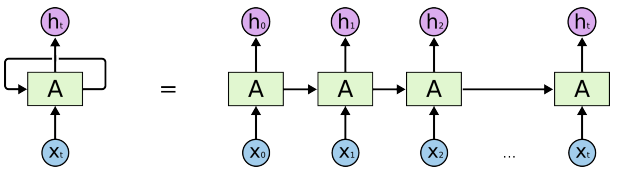


Figure 19: Unrolled representation of a RNN. Source : Understanding LSTM Networks by Christopher Olah - http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Let us mention the historical Simple Recurrent Networks introduced by Elman (1990) [4] and Jordan (1990) [7]. The recurrent network considered by Elman is a MLP with one unit layer looped back on itself. If we denote by $x(t)$ the input at time t, $\hat{y}(t)$ the output at time t and $\hat{z}(t)$ the hidden layer at time t, the model writes for the kth component of the output

$$\begin{aligned}\hat{y}^{(k)}(t) &= \sum_{i=1}^{I} W_i^{(k)} \hat{z}_i(t) + b^{(k)} \\ \hat{z}_i(t) &= \sigma\left(\sum_{j=1}^{J} w_{i,j} x_j(t) + \sum_{l=1}^{I} \tilde{w}_{i,l} \hat{z}_l(t-1) + b_i\right)\end{aligned}$$

where σ is an activation function. The neurons of the hidden layer that are looped to themselves are called *context units*. In the model introduced by Jordan, $\hat{z}_l(t-1)$ is replaced in the last equation by $\hat{y}_l(t-1)$. In this case, the *context units* are the output neurons. These models have been introduced in linguistic analysis. They are widely used in natural language processing. Nevertheless, the basic version of recurrent neural networks falls to learn long time dependancies. New architectures have been introduced to tackle this problem.

4.1 Long Short-Term Memory

In the last years, RNN have been successfully used again for various applications such as speech recognition, translation, image captioning .. This success is mostly due to the performances of LSTMs : Long Short-Term Memorys, which is a special kind of recurrent neural networks. Long Short-Term Memory (LSTM) cells were introduced by Hochreiter and Schmidhuber (1997) [10] and were created in order to be able to learn long time dependancies. A LSTM cell comprises at time t , a state C_t and an output h_t. As input, this cell at time t comprises x_t, C_{t-1} and h_{t-1}. Inside the LSTM, the computations are defined by doors that allow or not the transmission of information. These computations are governed by the following equations described in [10] .

$u_t = \sigma(W^u h_{t-1} + I^u x_t + b^u)$	Update gate H
$f_t = \sigma(W^f h_{t-1} + I^f x_t + b^f)$	Forget gate H
$\tilde{C}_t = \tanh(W^c h_{t-1} + I^c x_t + b^c)$	Cell candidate H
$C_t = f_t \odot C_{t-1} + u_t \odot \tilde{C}_t$	Cell output H
$o_t = \sigma(W^o h_{t-1} + I^o x_t + b^o)$	Output gate H
$h_t = o_t \odot \tanh(C_t)$	Hidden output H
$y_t = \text{softmax}(W h_t + b)$	Output K
W^u, W^f, W^c, W^o	Recurrent weights $H \times H$
I^u, I^f, I^c, I^o	Input weights $N \times H$
b^u, b^f, b^c, b^o	Biases H

Figure 20 reveals the main difference between a classical RNN and an LSTM. For a standard RNN, the repeated module A is very simple, it contains a single layer. For the LSTM, the repeated module contains four layers (the yellow boxes), interacting as described by the above equations.

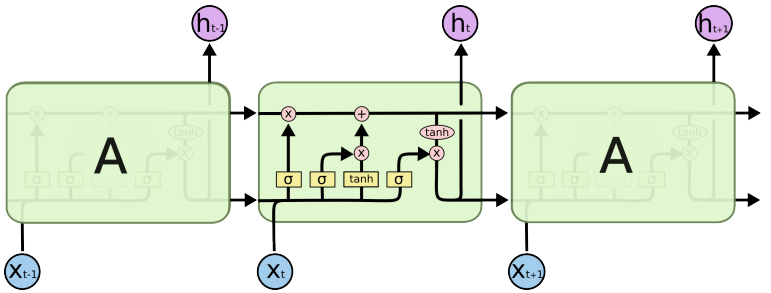


Figure 20: Diagram of an LSTM. Source : Understanding LSTM Networks by Christopher Olah http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-chain.png

Exercise. — On Figure 20, put the different elements mentioned in the equations defining the LSTM.

There are also variants of LSTMs, and this field of research is still very active to obtain more and more powerful models.

References

[1] Szegedy C., Ioffe S., Vanhouche V., and Alemi A. Inception-v4, inception-resnet and the impact of residual connections on learning. *Arxiv*, 1602.07261, 2016.

[2] Hinton G.E., Srivastava N., Krizhevsky A., Sutskever I., and Salakhutdinov R. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR, abs/1207.0580*, 2012.

[3] T. Hastie, R. Tibshirani, and J Friedman. *The elements of statistical learning : data mining, inference, and prediction*. Springer, 2009. Second edition.

[4] Elman J.L. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[5] D. Kingma and J. Ba. Adam : a method for stochastic optimization. *Arxiv*, 1412.6980, 2014.

[6] A. Krizhevsky, I. Sutskever, and G.E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, pages 1097–1105, 2012.

[7] Jordan M.I. *Artificial Neural Network, pages 112-127*. IEEE Press, 1990.

[8] Y. Nesterov. A method of solving a complex programming problem with convergence rate o(1/k2). *Soviet Mathematics Doklady*, 27:372–376, 1983.

[9] B.T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[10] Hochreiter S. and SchmidhuberJ. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[11] I. Sutskever, J. Martens, G.E. Dahl, and G.E. Hinton. On the importance of initialization and momentum in deep learning. *ICML*, 28(3):1139–1147, 2013.

[12] LeCun Y., Bottou L., Bengio Y., and Haffner P. Gradient-based learning applied to document recognition. *IEEE Communications magazine*, 27(11):41–46, 1998.

[13] LeCun Y., Jackel L., Boser B., Denker J., Graf H., Guyon I., Henderson D., Howard R., and Hubbard W. Handwritten digit recognition : Applications of neural networks chipsand automatic learning. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.