Institut de Mathématiques de Toulouse, INSA Toulouse

Neural Networks and Introduction to Deep Learning

High Dimensional and Deep Learning
INSA- Applied Mathematics

Béatrice Laurent

Outline

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.

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.

Introduction

- Deep learning architectures 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.
- 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).

Outline

Neural networks

- An artificial neural network is non linear with respect to its parameters θ that associates to an entry x an output $y = f(x, \theta)$.
- The neural networks can be use for regression or classification.
- 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).
- 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.

Artificial Neuron

Artificial neuron

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

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

Activation functions

Several activation functions can be considered.

Activation functions

- The identity function $\phi(x) = x$
- The sigmoid function (or logistic) $\phi(x) = \frac{1}{1+e^{-x}}$
- The hyperbolic tangent function ("tanh")

$$\phi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- The hard threshold function $\phi_\beta(x) = \mathbb{1}_{x \geq \beta}$
- The Rectified Linear Unit (ReLU) activation function $\phi(x) = \max(0, x)$

Activation functions

The following Figure represents the activation function described above.

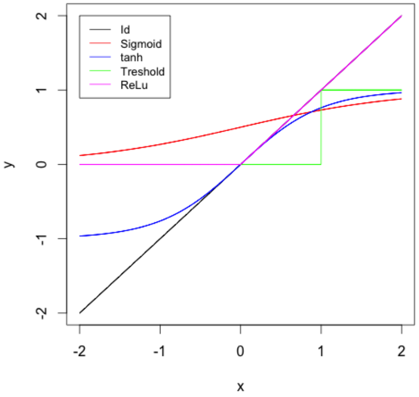
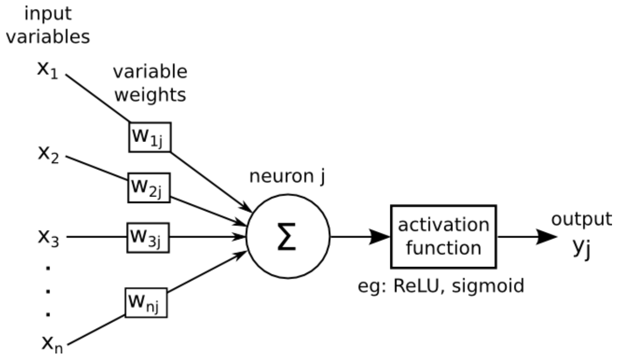


FIGURE: Activation functions

Artificial Neuron

Schematic representation of an artificial neuron where $\Sigma = \langle w_j, x \rangle + b_j$.



- 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.

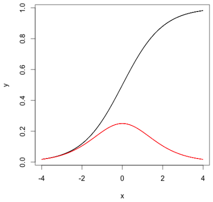


FIGURE: Sigmoid function (in black) and its derivatives (in red)

Activation functions

- With neural networks with a high number of layers, this causes troubles for the backpropagation algorithm to estimate the parameters.
- This is why the sigmoid function was supplanted by the rectified linear function (ReLu). This function is piecewise linear and has many of the properties that make linear model easy to optimize with gradient-based methods.
- The ReLU function and its derivative are equal to 0 for negative values, hence it is sometimes advised to replace it by

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

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

Outline

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.
- 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.

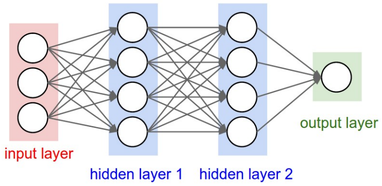


FIGURE: A basic neural network.

Multilayer perceptron

- 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.

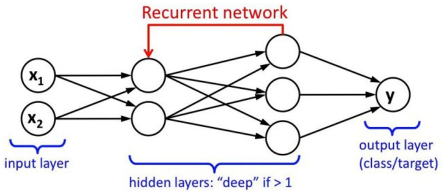


FIGURE: A recurrent neural network

Multilayers perceptrons

- 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. On the output layer, we apply no activation function (the identity function) in the case of regression.
- 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

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

Mathematical formulation

Multilayer perceptron with L hidden layers :

- We set $h^{(0)}(x) = x$.
  For $k = 1, \ldots, L$ (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
- 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 $k-1$.

Multilayer perceptron : a simple example

- Let us consider the XOR ("exclusive or") function defined for two binary inputs $x = (x_1, x_2) \in \mathbb{X} = \{0, 1\}^2$ by

$$\begin{aligned} f^*(x) &= 1 \text{ if } x = (0,1) \text{ or } (1,0) \\ &= 0 \text{ otherwise} \end{aligned}$$

- Here, we know exactly f^* on its whole definition set $\mathbb{X}$.
- We consider a parametric model $(x, \theta) \mapsto f(x, \theta)$ to approximate the function f^*.
- θ is obtained by minimizing the least square criterion

$$J(\theta) = \frac{1}{4} \sum_{x \in \mathbb{X}} (f^*(x) - f(x, \theta))^2.$$

Multilayer perceptron : a simple example

- We first consider a linear model defined for $\theta = (w, b)$ by
$$f(x, \theta) = \langle w, x \rangle + b, \quad w \in \mathbb{R}^2, b \in \mathbb{R}.$$
- We can easily show that the value of θ^* minimizing $\theta \mapsto J(\theta)$ is $\theta^* = ((0, 0), 1/2)$, which corresponds to $f(x, \theta^*) = 1/2\ \forall x \in \mathbb{X}$. (in exercice). Hence the XOR function cannot be well approximated by a linear function.
- On the other side, it can be perfectly represented by a perceptron with one hidden layer containing two units, with the ReLu activation function φ for the hidden layer and the identity function for the output layer. For all $x \in \mathbb{X}$,
$$f^*(x) = f(x, W, c, w, b) = w^T \varphi(W^T x + c) + b,$$

with
$$W = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad c = \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \quad w = \begin{pmatrix} 1 \\ -2 \end{pmatrix}, \quad b = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

Universal approximation theorem

Theorem

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

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

Universal approximation theorem

- This theorem due to Hornik (1991) shows 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 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.

Outline

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, generally with a gradient descent algorithm.
- We first have to choose the loss function.
- It is classical to estimate the parameters by minimizing 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

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

Loss functions

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

Loss functions

- For a multi-class classification problem, we consider a generalization of the previous loss function to k classes

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

- 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).

Penalized empirical risk

- The expected loss can be written as

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

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

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

Penalized empirical risk

- We can consider $\mathbb{L}^2$ regularization.

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

  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 $\Omega(\theta)$:

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

- One can also consider $\mathbb{L}^1$ regularization (LASSO penalty), leading to parcimonious solutions :

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

Penalized empirical risk

- In order to minimize the criterion $L_n(\theta)$, a **stochastic gradient descent algorithm** is often used.
- In order to compute the gradient, a clever method, called **Backpropagation algorithm** is considered.

The stochastic gradient descent algorithm

- Initialization of $\theta = (W^{(1)}, b^{(1)}, \ldots, W^{(L+1)}, b^{(L+1)})$.
- **For** $j = 1, \ldots, N$ iterations :
  - At step j :

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

where B is a subset of $\{1, \ldots, n\}$ with cardinality m.

The stochastic gradient descent algorithm

- In the previous algorithm, we do not compute the gradient for the empirical 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).

To apply the SGD algorithm, we need to compute the gradients $\bigtriangledown_\theta \ell(f(X_i, \theta), Y_i)$. For this, we use the Backpropagation algorithms.

Outline

Backpropagation algorithm for classification with the cross entropy

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

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

Backpropagation algorithm for classification with the cross entropy

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

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

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

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

for the loss function ℓ associated to the cross-entropy.

Backpropagation algorithm for classification with the cross entropy

Using the previous notations, 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 $z(x) = \phi(a_1(x), \dots, a_J(x))$, 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, using $f(x) = \text{softmax}(a^{(L+1)}(x))$, we have

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

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

$$\frac{\partial \ell(f(x), y)}{\partial (a^{(L+1)}(x))_i} = -\sum_j \frac{\mathbb{1}_{y=j}}{(f(x))_y} \frac{\partial \mathrm{softmax}(a^{(L+1)}(x))_j}{\partial (a^{(L+1)}(x))_i}$$

$$= -\frac{1}{(f(x))_y} \frac{\partial \mathrm{softmax}(a^{(L+1)}(x))_y}{\partial (a^{(L+1)}(x))_i}$$

$$= -\frac{1}{(f(x))_y} \mathrm{softmax}(a^{(L+1)}(x))_y (1 - \mathrm{softmax}(a^{(L+1)}(x))_y) \mathbb{1} y = i$$

$$+ \frac{1}{(f(x))_y} \mathrm{softmax}(a^{(L+1)}(x))_i \mathrm{softmax}(a^{(L+1)}(x))_y \mathbb{1}_{y \neq i}$$

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

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 $\mathbb{1}_{i=y}$.
We now obtain easily the partial derivative of the loss function with respect to the output bias. Since

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

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

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

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

Hence

$$\nabla_{W^{(L+1)}} \ell(f(x),y) = (f(x)-e(y))(h^{(L)}(x))'. \tag{2}$$

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

$$\nabla_{W^{(k)}}\ell(f(x),y) = \nabla_{a^{(k)}(x)}\ell(f(x),y)h^{(k-1)}(x)'. \tag{3}$$

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

$$\nabla_{b^{(k)}}\ell(f(x),y) = \nabla_{a^{(k)}(x)}\ell(f(x),y). \tag{4}$$

Backpropagation algorithm

We can now summarize the backpropagation algorithm.
We use the Backpropagation equations to compute the gradient by a two pass algorithm.

- **Forward pass :**
  - We fix the value of the current weights $\theta^{(r)} = (W^{(1,r)}, b^{(1,r)}, \ldots, W^{(L+1,r)}, b^{(L+1,r)})$,
  - 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.

- **Backward pass :** For $(x, y) = (X_i, Y_i)$, i in the batch B,
  - 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

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

Backpropagation algorithm

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

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

    where B is a batch with cardinality m.
  - Update the parameters

$$\theta^{(r+1)} = \theta^{(r)} - \varepsilon_r \tilde{\nabla}_\theta,$$

    where $\varepsilon_r > 0$ is the learning rate.
- 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.

Backpropagation algorithm for the regression

- In the regression case, where the empirical loss function is

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

with

$$\ell(f(X_i, \theta), Y_i) = \|Y_i - f(X_i, \theta)\|^2,$$

use the previous computations (in the classification case) to write the backpropagation equations.

Outline

Optimization algorithms

- Many algorithms can be used to minimize the loss function with various hyperparameters to calibrate.
- The Stochastic Gradient Descent (SGD) algorithm is commonly used for (deep) neural networks :

$$\theta^{(r+1)} = \theta^{(r)} - \varepsilon_r \frac{1}{m} \sum_{i \in B} \nabla_\theta \ell(f(X_i, \theta), Y_i),$$

where ε_r 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_r, and reduce this value during the successive iterations.The observation of the evolution of the loss function can give indications on the way to proceed.

Optimization algorithms

- A stopping rule, called *early stopping* is also often used : we stop learning when the loss function for a validation sample stops to decrease.
- *Batch learning* is used for computational reasons, backpropagation algorithms need to store intermediate values : 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.

SGD algorithm

Summary of 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

$$g \leftarrow \frac{1}{m} \sum_{i \in B_l} \nabla_\theta \ell(f(X_i, \theta), Y_i).$$

  - Update the parameter : $\theta \leftarrow \theta - \varepsilon g$.

SGD algorithm

- Since the choice of the learning rate is delicate, variations of the SGD algorithm that are less sensitive to this parameter have been proposed.
- **Stochastic gradient descent with momentum** and **Nesterov accelerated gradient** : Nesterov (1983) and Sutskever et al. (2013).
- **RMSProp** algorithm, due to Hinton (2012) .
- **Adam** algorithm, Kingma and Ba (2014).

SGD with momentum

- SGD may be very slow to converge. The momentum algorithm (Polyak, 1964) allows to accelerate learning.
- The idea is to accumulate an exponentially decaying moving average of past negative gradients and to continue to move in their direction.
- The momentum algorithm introduces a variable ν, that plays the role of a velocity.
- An hyperparameter $\alpha \in [0, 1[$ determines how fast the contribution of previous gradients exponentially decay.

SGD with momentum

The algorithm can be described as follows :

Stochastic Gradient Descent algorithm with momentum

- Fix the parameters : learning rate ε, momentum parameter $\alpha \in [0,1[$.
- Choose the initial parameter θ and the initial velocity ν.
- **While** stopping criterion not met **do**
  - Sample a minibach B of size m from the learning sample.
  - Compute the gradient estimate : $g \leftarrow \frac{1}{m}\sum_{i\in B} \nabla_\theta \ell(f(X_i,\theta), Y_i)$.
  - Update the velocity : $\nu \leftarrow \alpha\nu - \varepsilon g$.
  - Update the parameter : $\theta \leftarrow \theta + \nu$.

end while

SGD with Nesterov momentum

- Sutskever et al (2013) propose a variant of the momentum algorithm inspired by Nesterov accelerated gradient method (Nesterov, 1983).
- The variants lye in the updates of the parameter and the velocity :

$$\nu \leftarrow \alpha\nu - \varepsilon \frac{1}{m} \sum_{i \in B} \nabla_\theta \ell(f(X_i, \theta + \alpha\nu), Y_i)$$
$$\theta \leftarrow \theta + \nu.$$

- The learning rate ε is a difficult parameter to calibrate because it significantly affects the performances of the neural network. This is why new algorithms (RMSProp and Adam) have been introduced, to be less sensitive to this learning rate.

The RMSProp algorithm

- The idea of the RMSProp algorithm (Hinton, 2012) is to use a different learning rate for each parameter (components of θ) and to automatically adapt this learning rate during the training.
  The algorithm is described as follows.

RMSProp algorithm

```
• Fix the parameters : Global learning rate $\varepsilon$, decay rate $\rho$ in $[0,1[$.
• Choose the initial parameter $\theta$.
• Choose a small constant $\delta$, usually $10^{-6}$ (to avoid division by 0).
• Initialize accumulation variable $r = 0$.
• While stopping criterion not met do
    • Sample a minibach $B$ of size $m$ from the learning sample.
    • Compute the gradient estimate : $g \leftarrow \frac{1}{m}\sum_{i\in B}\nabla_\theta \ell(f(X_i,\theta),Y_i)$.
    • Accumulate squared gradient $r \leftarrow \rho r + (1-\rho) g \odot g$
    • Update the parameter : $\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\delta + r}} \odot g$ ($\frac{1}{\sqrt{\delta + r}}$ is computed element-wise).
end while
```

Adam algorithm

- Adam algorithm (Kingma and Ba, 2014) is also an adaptive learning rate optimization algorithm.
- "Adam" means "Adaptive moments".
- It can be viewed as a variant of RMSProp algorithm with momentum.
- It also includes a bias correction of the first order moment (momentum term) and second order moment.

Adam algorithm

Adam algorithm

- Fix the parameters : Global learning rate ε, decay rate for moment estimates ρ_1 and ρ_2 in $[0, 1[$.
- Choose the initial parameter θ.
- Choose a small constant δ, usually 10^{-8} (to avoid division by 0).
- Initialize 1st and 2nd moment variables variable $s = 0$, $r = 0$. Initial time step $t = 0$.
- **While** stopping criterion not met **do**

```
    Sample a minibach $B$ of size $m$ from the learning sample.
    Compute the gradient estimate : $g \leftarrow \frac{1}{m} \sum_{i \in B} \nabla_\theta \ell(f(X_i, \theta), Y_i)$.
    $t \leftarrow t + 1$
    Update first moment estimate $s \leftarrow \rho_1 s + (1 - \rho_1) g$
    Update second moment estimate $r \leftarrow \rho_2 r + (1 - \rho_2) g \odot g$
    Correct biases : $\hat{s} \leftarrow s/(1 - \rho_1^t)$, $\hat{r} \leftarrow r/(1 - \rho_2^t)$
    Update the parameter : $\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\hat{r} + \delta}} \odot \hat{s}$
end while
```

Optimization algorithm

- We have presented the most popular optimization algorithms for deep learning.
- There is actually no theoretical foundation on the performances of these algorithms, even for convex functions (which is not the case in deep learning problems !).
- Numerical studies have been performed to compare a large number of optimization algorithms for various learning problems (Schaul et al. (2014)). There is no algorithms that outperforms the other ones. The algorithms with adaptive learning seem more robust to the hyperparameters.
- The choice of the initialization of the parameter θ is also an important point.

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** introduced by Hinton et al. (2012).
- With a certain probability p, and independently of the others, each unit of the network is set to 0.
- It is classical to set p 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.

Dropout

- 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).

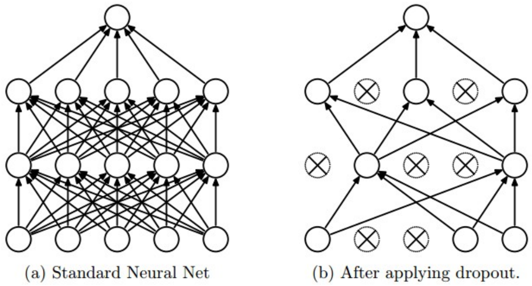


FIGURE: Dropout

Conclusion

- We have presented in this course the feedforward neural networks, and explained how the parameters of these models can be estimated.
- The choice of the architecture of the network is also a crucial point. Several models can be compared by using a cross validation method to select the "best" model.
- The perceptrons are defined for vectors. They are not well adapted for some types of data such as images. By transforming an image into a vector, we loose spatial information, such as forms.
- The **convolutional neural networks (CNN)** introduced by LeCun (1998) have revolutionized image processing.
- CNN act directly on matrices, or even on tensors for images with three RGB color channels.
- They are also based on the methods presented in this course to estimate their parameters (backpropagation equations, optimization algorithms ..)
- The presentation of the convolutional neural networks will be the topic of the next course.