Ecole Centrale Paris

Innovation Research Project - P5L237

Final Report

Translated from French by Hamza TAZI BOUARDI

A few Learning Algorithms for Deep Neural Networks

Authors:
Hamza Tazi Bouardi
Younes Belkouchi

Supervisor:
Paul-Henry Cournede

June 4th, 2018

Contents

1 Introduction

Deep Neural Networks are increasingly used in both industry and research in order to answer to a huge variety of problems. For example, they are used to classify complex images, for Natural Language Processing (NLP), or even in the analysis and data mining of genomic information. Initially, those Neural Networks aimed at imitating human thought, notably inspired by how information is processed in the human brain: electric signals flowing through synapses (which will be, in our Artificial Neural Network, the links between the different nodes).

Those Neural Networks have gained momentum these last few years, particularly thanks to their efficiency and their accuracy in predictions, as well as the numerous important breakthroughs in terms of, for example, their architectures (we are not stuck in a simple −but nonetheless efficient− Multilayer Perceptron anymore) or Optimization and Learning Algorithms (other than those covered in this study, such as **Adagrad** [Duchi et al., 2011] or **Adam** (Adaptive Moment Estimation) [Kingma et al., 2015]).

Let us also mention that this Neural Networks '*trend*' has been going along with a certain mystification of their concepts and implementation, especially given the fact that many libraries are very well implemented and ready to use as black boxes. One of the goals of this study in to show that these Neural Networks are simply another type of Non-Linear Learning Algorithms based on rather simple Optimizations Methods −**Classic** and **Stochastic Gradient Descent**− that will be studied and implemented.

Furthermore, this research project also has the ambition to understand the origins of Overfitting problems in Deep Neural Networks and to implement a rather recent and extremely efficient method [Srivastava et al., 2014] proposed by a researcher at the University of Toronto. Thus, a few questions naturally arise from this little introduction:

Among the Optimization Methods mentioned earlier, which one is the most efficient ?
How do they differ from each other ?
What is Overfitting and how can we solve the problems it generates ?

In this study of Learning Algorithms for Deep Neural Networks, we will first study the theoretical and mathematical foundations of Classic Gradient Descent and Backpropagation, as well as those of Stochastic and Mini-Batches Gradient Descent, which form the basis of Optimization of Neural Networks. We will then implement Deep Neural Networks without using any existing library in *Python* (that is to say, only *NumPy*) based on these very Optimization Methods in order to study and compare their respective performances. This comparative study will be done firstly on a very simple input −the XOR Function− to see if the Algorithms run properly, and then on a extremely famous dataset called '*Iris Dataset*', where the goal will be to classify flowers into 3 categories, as we will see in Section 3.3.2. Finally, we will study the Dropout Method, which aims at solving Overfitting problems often encountered with Deep Neural Networks.

2 Gradient Descent Algorithm

2.1 Gradient Descent - General Principle

The goal of this Optimization algorithm [Tibshirani, 2015] is to minimize a certain function f that we call '*Cost Function*'. We thus try to determine $x^* = \text{argmin } f(x)$ where f is *a priori* assumed convex and differentiable. From there, we can apply Taylor's theorem to our function to the 1^{st} order [Ciuperca, 2013]. So for $x_0 \in \mathbb{R}^n$, $t > 0$ and $n \in \mathbb{N}^*$ we have:

$$f(x_0 - t\nabla f(x_0)) \approx f(x_0) - t\nabla f(x_0)^T \nabla f(x_0) \tag{1}$$

We can see that we actually get closer to the minimum thanks to our function's slope, that is to say its gradient, since the value on the left is smaller than $f(x_0)$. We could also push the expansion to the 2^{nd} order, in order to obtain:

$$f(x_0 - t\nabla f(x_0)) \approx f(x_0) - t\nabla f(x_0)^T \nabla f(x_0) + \frac{1}{2}t^2 \nabla f(x_0)^T H_f(x_0) \nabla f(x_0) \tag{2}$$

Where H_f is the Hessian Matrix of f. The third term is actually a correction of this 'gradient descent' that could be credited to the 'curvature' of our function f. However, computing this matrix is often very costly in terms of both time and space complexity, so we will restrict our study to 1^{st} order optimization. To that end, we propose the following algorithm to obtain x^* starting from the observation above, given that the term $-t\nabla f(x)$ allows to descend 'on the right side'.

Classic Gradient Descent Algorithm:

1. Initialize at $x_0 \in \mathbb{R}^n$.
2. $\forall k \geq 1, x_k = x_{k-1} - t\nabla f(x_{k-1})$ where t is the 'size of the jumps', also known as '*Learning Rate*' in Deep Neural Networks' lingo.
3. Stop when we get to a minimum.

We also want to highlight the fact that we talk of '*a*' minimum (and not '*the*' minimum, meaning the global minimum). It is in this subtlety that resides the limit of Classic Gradient Descent. Indeed, as we will see in Section 2.2, the hypotheses assumed on our Cost Function are quite strong (especially the convexity), and the algorithm collapses if this hypothesis is not verified anymore (nothing guarantees the convergence towards a minimum, even a local one). We will see in Section 4.1 that there is a method allowing us to overcome the necessity of this type of hypotheses and still obtain at least a local minimum: it is called *Stochastic Gradient Descent*.

2.2 Gradient Descent - Proof of Convergence

Property 1: Let $f : \mathbb{R}^n \to \mathbb{R}$ be a convex and differentiable function with a L-Lipschitzian $(L > 0)$ *i.e.* such as:

$$\forall x, y \in \mathbb{R}^n, ||\nabla f(x) - \nabla f(y)||_2 \leq L||x - y||_2 \tag{3}$$

Then for k steps of Gradient Backpropagation, with a fixed size of jumps $t \leq \frac{1}{L}$, we have, for $f(x^*)$ the optimal value output by our Algorithm:

$$f(x_k) - f(x^*) \leq \frac{{||x_0 - x^*||_2}^2}{2tk} \tag{4}$$

Which results in a convergence in $O(1/k)$.

Lemma 1: Under the hypotheses of Property 1, we have:

$$\nabla^2 f(x) \preceq LI \text{ which is equivalent to } \nabla^2 f(x) - LI \text{ Negative Semi-Definite} \tag{5}$$

Proof Lemma 1:

$$\begin{aligned} f \text{ convex with a } L\text{-lipschitzian gradient} &\Leftrightarrow \phi(x) = \frac{L}{2}x^T x - f(x) \text{ convex} \\ &\Leftrightarrow \nabla^2 \phi(x) \succeq 0 \\ &\Leftrightarrow LI \succeq \nabla^2 f(x) \end{aligned}$$

Preuve Propriété 1: Using Taylor's Theorem, we have for $x, y \in \mathbb{R}^n$:

$$\begin{aligned} f(y) &\approx f(x) + \nabla f(x)^T(y - x) + \frac{1}{2}\langle \nabla^2 f(x)(y - x), y - x\rangle \\ &\leq f(x) + \nabla f(x)^T(y - x) + \frac{1}{2}|||\nabla^2 f(x)||| \cdot ||y - x||_2^2 \\ &\leq f(x) + \nabla f(x)^T(y - x) + \frac{1}{2}|||LI||| \cdot ||y - x||_2^2 \text{ given (5)} \\ &\leq f(x) + \nabla f(x)^T(y - x) + \frac{L}{2}||y - x||_2^2 \text{ because } |||I||| = 1 \end{aligned}$$

Given the fact that $|||A||| = \sup_{x \in \mathbb{R}^n \setminus \{0\}} \frac{||Ax||}{||x||}$. Therefore, for $\boxed{y = x^+ = x - t\nabla f(x)}$:

$$\begin{aligned} f(x^+) &\leq f(x) - t||\nabla f(x)||_2^2 + \frac{Lt^2}{2}||\nabla f(x)||_2^2 \\ &\leq f(x) - t(1 - \frac{Lt}{2})||\nabla f(x)||_2^2 \\ &\leq f(x) - \frac{t}{2}||\nabla f(x)||_2^2 \; \textit{since } t \leq \frac{1}{L} \end{aligned}$$

On the other hand, given the convexity of f (which can be found with Taylor's Theorem and Equation (2) since the Hessian matrix is Positive Definite so the third term of the expansion is positive):

$$f(x^*) \geq f(x) + \nabla f(x)^T(x^* - x) \Leftrightarrow f(x) \leq f(x^*) + \nabla f(x)^T(x - x^*)$$

So by injecting in the latest equation we obtain:

$$\begin{aligned}
&f(x^+) \leq f(x^*) + \nabla f(x)^T(x - x^*) - \frac{t}{2}||\nabla f(x)||_2^2 \\
\Leftrightarrow &f(x^+) - f(x^*) \leq \frac{1}{2t}(||x - x^*||_2^2 - (t^2||\nabla f(x)||_2^2 + ||x - x^*||_2^2 - 2t\nabla f(x)^T||x - x^*||_2^2)) \\
\Leftrightarrow &f(x^+) - f(x^*) \leq \frac{1}{2t}(||x - x^*||_2^2 - ||x - t\nabla f(x) - x^*||_2^2) \\
\Leftrightarrow &f(x^+) - f(x^*) \leq \frac{1}{2t}(||x - x^*||_2^2 - ||x^+ - x^*||_2^2)
\end{aligned}$$

We then sum over the k steps of our Classic Gradient Backpropagation Algorithm and obtain:

$$\begin{aligned}
\sum_{i=1}^{k}(f(x_i) - f(x^*)) &\leq \frac{1}{2t}\sum_{i=1}^{k}(||x_{i-1} - x^*||_2^2 - ||x_i - x^*||_2^2) \\
&\leq \frac{1}{2t}(||x_0 - x^*||_2^2 - ||x_k - x^*||_2^2) \\
&\leq \frac{||x_0 - x^*||_2^2}{2t} \text{ car retrait d'un terme positif.}
\end{aligned}$$

Finally, since the values taken by f diminish at each iteration of the Algorithm:

$$\boxed{f(x_k) - f(x^*) \leq \frac{1}{k}\sum_{i=1}^{k}(f(x_i) - f(x^*)) \leq \frac{||x_0 - x^*||_2^2}{2tk}}$$

Because $\forall i \in \{1, ..., k\},\ f(x_i) - f(x_*) \geq f(x_k) - f(x^*)$

3 Classic Gradient Backpropagation

3.1 Gradient Backpropagation - General Principle

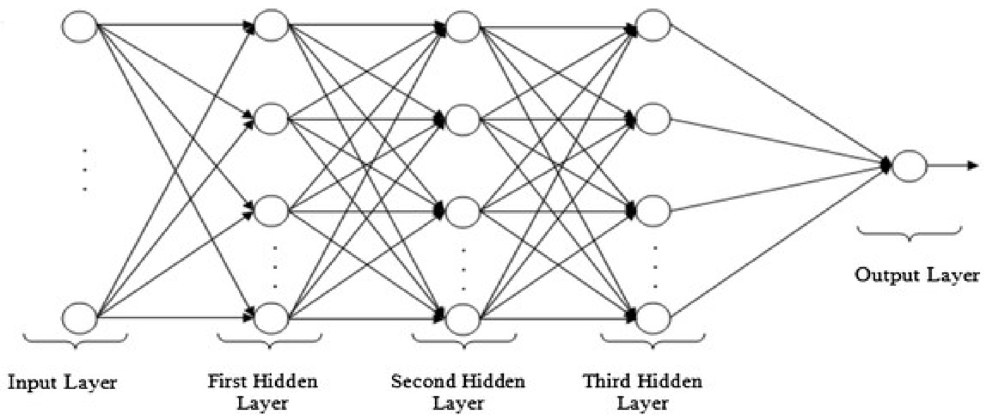


Figure 1: Bloc diagram of a MultiLayer Perceptron (MLP) with three hidden layers

First of all, let us highlight the fact that Gradient Descent is an Algorithm, while Gradient Backpropagation is another Algorithm that actually uses Gradient Descent as we will see later in this section. From now on, we will use the Multilayer Perceptron (MLP) represented in Figure 2 or Figure 4. Each connection between the different neurons is affected with a certain weight (taking its values in $[0, 1]$) that will be called,depending on the computations and the representations used, either w_{ij}, β_{ij} or even α_{ij} (for i, j in a certain specified partition of $\mathbb{N}$). The *inputs* (respectively *outputs*), on the other hand, will always be called X_i (respectively Y_i), and will be real numbers. Finally, we will call Z_i the vectorial intermediary values after linear combination of the *inputs* with the weights defined earlier (thus using a single-layered MLP for simplicity sakes, cf. Figure 4). Moreover, let us highlight the fact that a Neural Network Algorithm runs through two major steps:

1. A first step called *Forward Propagation*: it consists in a weighted linear combination (with the weights defined earlier characterizing the Neural Network) between each neuron, and the application, at each neuron, of an *Activation Function* that also help allow the learning (let us quote for instance the *Sigmoid Function*: $\sigma : x \mapsto \frac{1}{1+e^{-x}}$)

2. A second step called *Backward Propagation*: it consists in the computation (or an estimation) of a certain *Cost Function* (for instance *Mean Squared Error* or *Cross Entropy Loss*) that will depend on the Network's weights. The algorithm will then perform a Gradient Descent in order to obtain the weights that *minimize* this Error/Cost between our Predictions and the Optimal Values.

3. Repeat these two steps as many times as necessary to converge towards a solution.

This shows that a Neural Network is, after all, nothing but a non-linear Optimization Problem. We will now give a little more details about the theoretical foundations of a Neural Network and how those two steps happen in practice.

3.2 Gradient Backpropagation - Theoretical Foundations

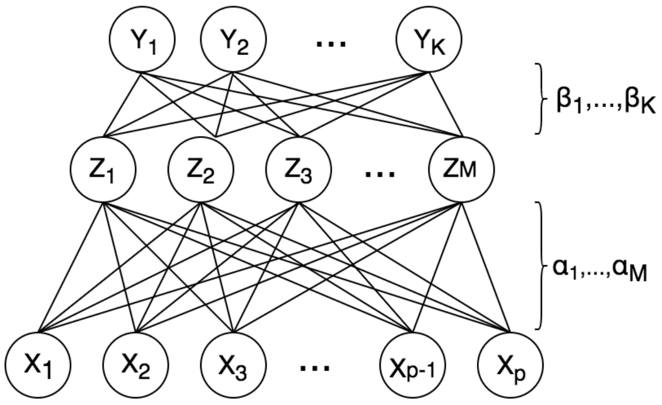


Figure 2: Bloc Diagram of the MLP used for computations

We will, from now on, use the Neural Network represented in Figure 2 in order to do the computations allowing the optimization of our coefficients and the application of the Gradient Descent Algorithm. This method [Tibshirani et al., 2017] consists first of all in defining the following variables:

$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X) \; \forall m \in \{1, ..., M\}, X = (X_1, ..., X_p) \in \mathbb{R}^{N\times p} \text{ and } \sigma \text{ the Activation Function.}$$
$$T_k = \beta_{0k} + \beta_k^T Z \quad \forall k \in \{1, ..., K\} \; and \; Z = (Z_1, ..., Z_M) \in \mathbb{R}^{N\times M} \; and \; T = (T_1, ..., T_K)$$
$$f_k(X) = g_k(T) = Y_k \quad \forall k \in \{1, ..., K\} \tag{6}$$

Therefore, we have $\forall m \in \{1, ..., M\}, \; \alpha_m \in \mathbb{R}^p$ and $\forall k \in \{1, ..., K\}, \; \beta_k \in \mathbb{R}^M$, and finally α_{0m} *and* β_{0k} corresponding to the intercepts (*cf.* Linear Regression). Furthermore, we define our *Cost Function* as follows:

$$\boxed{J = \sum_{k=1}^{K}\sum_{i=1}^{N}(Y_{ik} - f_k(x_i))^2 = \sum_{k=1}^{K}\sum_{i=1}^{N}\left(Y_{ik} - g_k(\beta_k^T z_i)\right)^2 = \sum_{i=1}^{N} J_i} \tag{7}$$

Deriving our Cost Function with respect to each of the two types of coefficients (in order to get the minimum), we use the Chain Rule, and we obtain the two following equations for $i \in \{1, ..., N\}$ and $l \in \{1, ..., p\}$ and $Z_{mi} = \sigma(\alpha_{0m} + \alpha_m^T X_i)$:

$$\begin{aligned}\frac{\partial J_i}{\partial \beta_{km}} &= \frac{\partial J_i}{\partial f_k(X_i)}\frac{\partial f_k(X_i)}{\partial \beta_{km}} \\ &= -2(Y_{ik} - f_k(X_i))g_k'(\beta_k^T Z_i) = \boxed{\delta_{ki} Z_{mi}} \; where \; Z_i = (Z_{1i}, ..., Z_{Mi})\end{aligned} \tag{8}$$

$$\begin{aligned}\frac{\partial J_i}{\partial \alpha_{ml}} &= \frac{\partial J_i}{\partial f_k(X_i)}\frac{\partial f_k(X_i)}{\partial Z_i}\frac{\partial Z_i}{\partial \alpha_{ml}} \\ &= -2\left(\sum_{k=1}^{K}(Y_{ik} - f_k(X_i))g_k'(\beta_k^T Z_i)\beta_{km}\right) X_{il}\sigma'(\alpha_m^T X_i) = \boxed{s_{mi} X_{il}}\end{aligned} \tag{9}$$

Therefore, s_{mi} corresponds to the error in the hidden layer (middle layer) and δ_{ki} to the error in the output layer (the highest in Figure 2). Given the definition of these errors, they verify the following equation called *'Backpropagation Equation'*:

$$\boxed{s_{mi} = \sigma'(\alpha_m^T X_i) \sum_{k=1}^{K} \beta_{km} \delta ki} \tag{10}$$

Thus, equation (6) allows us to compute the quantities necessary for the *Forward Propagation* step, while equations (10) allows the optimization of our weights/coefficients using the Gradient Descent Algorithm and therefore for the *Backward Propagation* step. Indeed, for each weight, and given a certain *Learning Rate* $t > 0$ (*cf.* Section 2.1), we obtain the new updated weights at each training step of the algorithm (that we will call epoch, which is equivalent to one forward and backward propagation on the whole dataset) defined as follows:

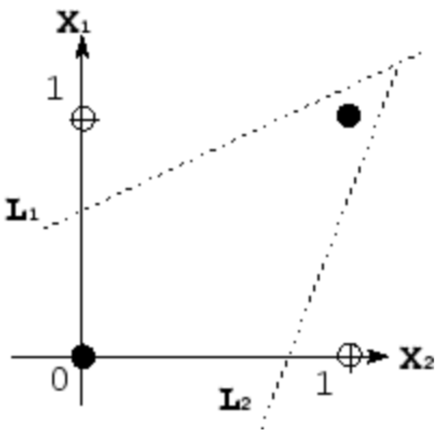
$$\boxed{\alpha_{ml}^* = \alpha_{ml} - t\frac{\partial J_i}{\partial \alpha_{ml}} = \alpha_{ml} - t s_{mi} X_{il}} \quad \boxed{\beta_{km}^* = \beta_{km} - t\frac{\partial J_i}{\partial \beta_{km}} = \beta_{km} - t\delta_{ki} Z_{mi}} \tag{11}$$

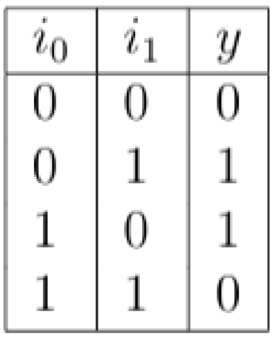
3.3 Gradient Backpropagation - Algorithm Implementation

For simplicity sakes (and to facilitate the reading of the Code in *Appendix A - Neural Network Code*) as well as for optimizing the computation time (and thus improve the efficiency of the Algorithm) we have implemented the Neural Network withe Matrices (instead of loops for instance), hence the change of notations from the previous Section.

3.3.1 Simple Case Study: XOR Function

Let's take the example of a Neural Network trying to learn solving the *Exclusive OR* function (also known as the *XOR* function). Given two inputs in $\{0,1\}$, the output (from the network) must verify the table in Figure 3 (defining the XOR function that goes from $\{0,1\}^2$ to $\{0,1\}$), while the very same Figure 3 shows a representation of the XOR function.





i_0	i_1	y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 3: (*Left*) XOR Function Graph (Source). (*Right*) XOR Function Table.

This problem cannot be solved with a simple Linear Regression, as it can be seen in the XOR Function Graph that two lines are necessary in order to separate the two categories of answers. We thus have decided to use a simple Neural Network composed of a single hidden layer with 2 neurons. Let us set:

$$\begin{aligned}
& i = x_0 = \begin{bmatrix} i_0 \\ i_1 \end{bmatrix}, \\
& W_1 = \begin{bmatrix} w_{0,0}^{(1)} & w_{1,0}^{(1)} \\ w_{0,0}^{(1)} & w_{1,0}^{(1)} \end{bmatrix}, \; x_1 = \begin{bmatrix} a_{1,0} \\ a_{1,1} \end{bmatrix} = \begin{bmatrix} f(x_{1,0}) \\ f(x_{1,1}) \end{bmatrix}, \\
& W_2 = \begin{bmatrix} w_{0,0}^{(2)} & w_{0,1}^{(2)} \end{bmatrix}, \; x_2 = o = \begin{bmatrix} a_{2,0} \end{bmatrix} = \begin{bmatrix} f(x_{2,0}) \end{bmatrix}
\end{aligned} \tag{12}$$

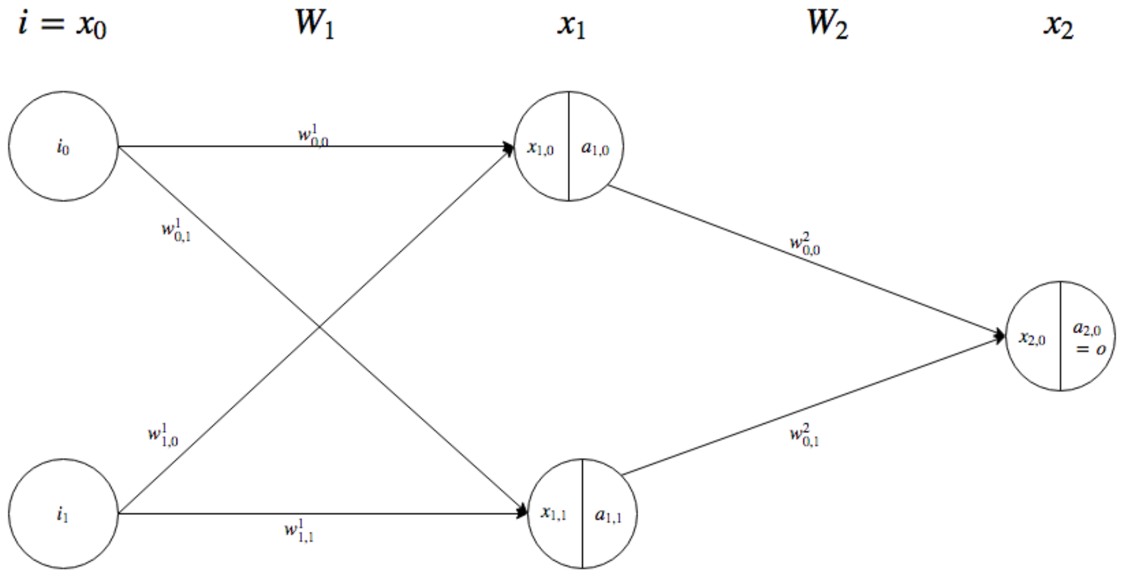


Figure 4: Neural Network performing a XOR function

Forward Propagation: A simple matrix computation leads us to see that:

$$\begin{cases} a_{1,0} = f(x_{1,0}) = f(w_{0,0}^{(1)} * i_0 + w_{1,0}^{(1)} * i_1) \\ a_{1,1} = f(x_{1,1}) = f(w_{0,1}^{(1)} * i_0 + w_{1,1}^{(1)} * i_1) \end{cases} \Leftrightarrow x_1 = W_1 x_0 \tag{13}$$

Same thing goes for:

$$a_{2,0} = f(x_{2,0}) = f(w_{0,0}^{(2)} * a_{1,0} + w_{0,1}^{(2)} * a_{1,1}) \Leftrightarrow x_2 = W_2 x_1 \tag{14}$$

Error Computation: Let us consider the Mean Squared Error for our problem. We obtain:

$$J(o) = \frac{1}{2}||Y - O||_2^2 = \frac{1}{2}(y - a_{2,0})^2 = \frac{1}{2}(y - o)^2 \tag{15}$$

The partial derivative of the latter with respect to the ouput is very simple:

$$\frac{\partial J}{\partial o} = -(y - o) \tag{16}$$

Gradient Backpropagation: To proceed, we will use what is called the 'Delta Method', which consists in computing a certain factor δ and to multiply it by the values of the node in order to fit the weights and obtain the new ones. It is used because it is very algorithmic and intuitive. Based on Equations (9) et (10), we can compute $\frac{\partial J}{\partial W_i}$, starting with the last layer:

$$\begin{aligned}
\frac{\partial J}{\partial W_2} &= \frac{\partial J}{\partial O}\frac{\partial O}{\partial W_2} \\
&= -(y - x_2).\frac{\partial f(W_2 x_1)}{\partial W_2} \\
&= -(y - x_2) \circ f'(W_2 x_1)\frac{\partial W_2 x_1}{\partial W_2} \\
&= -(y - x_2) \circ f'(W_2 x_1) x_1^T
\end{aligned} \tag{17}$$

Let us set $\delta_2 = -(y - x_2) \circ f'(W_2 x_1)$ and we obtain:

$$\boxed{\frac{\partial J}{\partial W_2} = \delta_2 x_1^T} \tag{18}$$

The computations are perfectly similar for the next layer:

$$\begin{aligned}
\frac{\partial J}{\partial W_1} &= \frac{\partial J}{\partial O}\frac{\partial O}{\partial W_1} \\
&= -(y - x_2)\frac{\partial f(W_2 x_1)}{\partial W_1} \\
&= -(y - x_2) \circ f'(W_2 x_1)\frac{\partial W_2 x_1}{\partial W_1} \\
&= \delta_2.\frac{\partial W_2 x_1}{\partial W_1} \\
&= W_2^T \delta_2.\frac{\partial f(W_1 x_0)}{\partial W_1} \\
&= W_2^T \delta_2 \circ f'(W_1 x_0)\frac{\partial (W_1 x_0)}{\partial W_1}
\end{aligned} \tag{19}$$

We set: $\delta_1 = [W_2^T \delta_2 \circ f'(W_1 x_0)]$
And we thus obtain:

$$\boxed{\frac{\partial J}{\partial W_1} = \delta_1 x_0^T} \tag{20}$$

Weight Fitting: We use Gradient Descent (detailed above) to fit the weights (*cf.* Gradient Descent - General Principle):

$$\boxed{W_i^{fit} = W_i - learning\_rate * \frac{\partial J}{\partial W_i}} \tag{21}$$

And we repeat these steps until our error converges (if everything goes as planned, towards a minimum).

3.3.2 General Case

The computations are as follows:
<u>Forward Propagation</u>:

$$\begin{aligned} inputs = x_0,\ x_1 = f(W_1 x_0), \\ \forall i \in \{1,..,n\},\ x_i = f(W_i x_{i-1}), \\ outputs = x_n = f(W_n x_{n-1}) \end{aligned} \tag{22}$$

<u>Backward Propagation</u>: Gradient Descent in the general case is very similar to the one in the XOR Case. For the last (output) layer:

$$\boxed{\frac{\partial E}{\partial W_n} = \delta_n . x_{n-1}^T \text{ avec } \delta_n = \frac{\partial E}{\partial x_n} \circ f'(W_n x_{n-1})} \tag{23}$$

For the other layers:

$$\boxed{\frac{\partial E}{\partial W_i} = \delta_i . x_{i-1}^T \text{ avec } \delta_i = W_{i+1}^T \frac{\partial E}{\partial x_i} \circ f'(W_i x_{i-1})} \tag{24}$$

Weight fitting is done following the equation below (*cf.* Gradient Descent - General Principle):

$$\boxed{W_i^{ajust} = W_i - learning\_rate * \frac{\partial E}{\partial W_i}} \tag{25}$$

4 Stochastic Gradient Descent

4.1 General Principle

Up until now, we have always used the whole dataset as an input to the algorithm and for each epoch (*i.e.* forward and backward propagation of the network), especially for the computation of the cost, thus using each and every example in our dataset to have an exact value of the cost function at each training step. However, in the case of greater scale applications (in terms of the size of dataset, for instance data on searches made by Google users) which could contain millions or even billions of entries and examples. Therefore, it seems obvious that in these cases, computing the cost on the whole dataset can become extremely costly, both in terms of time and space (data and results storage).

Thus, many optimization algorithms [Goodfellow et al., 2016] based on gradient computations are much more efficient (in terms of convergence time) when they compute a simple estimation of the gradient instead of the exact gradient of the Cost function (and therefore an approximate value of the optimal Cost). This is all the more relevant in the case of Complex Neural Networks where we try to do Statistical Learning. Indeed, given the redundancy and similarity that characterize most datasets on which we will be working in the real life, it is perfectly justified to use only small data 'packets' − which we will call 'mini-batches' in the sequel − for the computation of the gradient, especially since many examples will have a very similar contribution to a given computation, and this considerably reduces convergence times in the case of large datasets.

The Stochastic Gradient Descent Algorithm uses the principles outlined above and is extremely efficient with large datasets. So, unlike the Classic Gradient Descent Algorithm − from which it directly comes −, the gradient computation of our Cost Function for the *Backpropagation* step in a simple estimation, as it is a computation using a mini-batch of data (the size being chosen) instead of all the data, which considerably reduces the necessary computation power at each step. A crucial point in this algorithm is the **random choice** of the elements of the batches to avoid any (additional) bias in the results and to preserve a certain 'independence' inside the data. Indeed, one could imagine a dataset in which the data that follow each other are very similar (for example blood groups of patients [Tibshirani et al., 2017]) and choosing those which follow each other would completely bias the outputs of the algorithm).

Moreover, it is this randomness that allows the Algorithm not to be stuck in local minima [Bottou, 1991] and find, unlike the classic algorithm, almost systematically a global minimum (or at the very least a very satisfying local minimum). It is also the randomness [De Sa and al, 2017] that allows us to get rid of very strong mathematical hypotheses (such as Cost Function convexity) and thus to treat a very wide range of new problems. Finally, the name '*Stochastic*' will be reserved for the cases in which each mini-batch contains one example only, while the '*Mini-Batch Gradient*' designation will be applicable to the general case. Our implementation will allow the user to use both, because each method can be more or less efficient, depending on the data and its characteristics (correlation, redundancy, similarity...).

4.2 Algorithm Implementation

We have decided to shuffle the input data of the Algorithm in order to randomly draw the mini-batches (of variable sizes, according to the choice of the user, which will be a function of the data itself as explained in the previous paragraph) which will be used for the estimation of the gradient at each epoch. There was no particular difficulty of implementation, since all we had to do was add to our Classic Retropropagation Neural Network a method doing what has been described above.

This is when we decided to use our Neural Network on a real dataset rather than on a simple *XOR* function, and we decided to use the famous *Iris Dataset* [Taniskidou and al, 2018]. It consists in a dataset of 150 examples of flowers (iris) provided by the University of California, Irvine. The goal is to classify these flowers into 3 classes – *Iris Setosa*, *Iris Versicolour* and *Iris Virginica* – bases on 4 attributes: length and width of sepals and petals.

We have had a few difficulties making the algorithm converge because we had noticed that we had, in a certain way, overfitted our algorithm to the XOR function, which raised many problems (as detailed before). The major difficulty has been to find the right Activation function (depending on the status of the neurons, *i.e.* if they were hidden or visible) as well as the right Cost function for this classification problem (since the Sum of Squares isn't adapted to this kind of problems). After studying the Sickit Learn documentation [ScikitLearn, 2017] as well as a few searches through the *StackOverFlow* community, we finally chose to use the *ReLu or Rectifier* activation function for the hidden neurons, a *Softmax* activation function for the terminal (and visible) neuron, and a *Cross Entropy* Cost function.

4.3 Comparison with Classic Gradient Descent

We then decided to compare the performances of both algorithms (Classic and Stochastic Backpropagation) on this very *Iris Dataset*, for which we have also determined the most suitable and performing hyperparameters, that is to say the optimal numbers of batches and epochs, as well as the optimal learning rate.

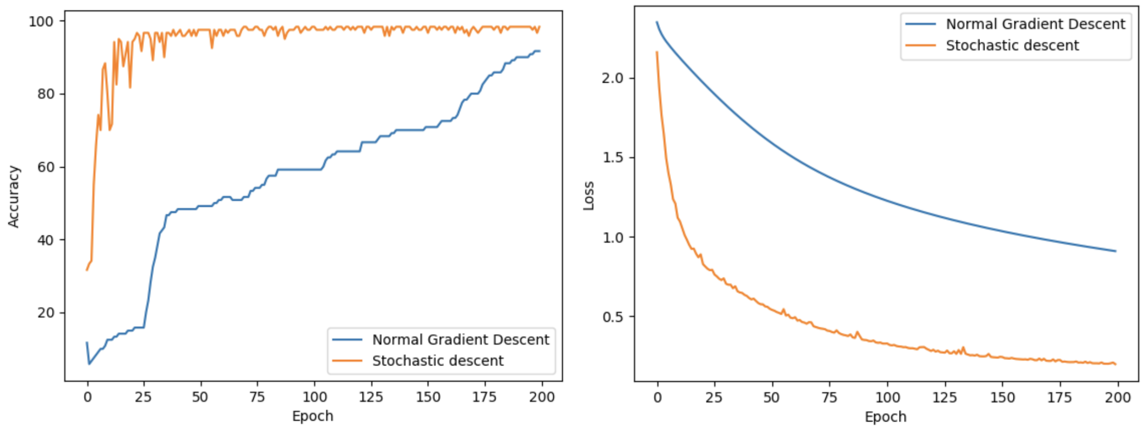


Figure 5: **(Left)** Accuracy against Number of Epochs
(Right) Cost function against Number of Epochs.

However, the reader must understand that the choice of these hyperparameters has been done successively, not by using a cross-validation (or similar) process because we wanted to implement every function used in this project (*i.e.* without using any existing library except *numpy*) and we didn't have time to do so. We are aware it would have been much better to choose all the parameters 'at the same time' by studying the influence of each one over each others, but our solution was more convenient. Therefore, we started by choosing the number of epochs (which we thought was the most important regarding calculational power), and we have improved the obtained number with the choice of the learning rate, and then with the choice of the number of batches. The plotted graphs of this section correspond to the results of our study, each one containing the optimal value for the hyperparameters.

Thus, we have plotted in Figure 5 the Accuracy and the value of the Cost function against the number of epochs. Both curves (e.g. for each algorithm) in each graph evolve exactly the same way, in the sense that the Accuracy increases with the number of epochs as the cost decreases in parallel, which makes perfect sense since we brows more and more times our dataset, giving the algorithm more opportunities to learn. Another very important detail to notice is the convergence speed of the Stochastic Gradient (for 120 batches here) compared to the one of Classic Gradient: the former allows an optimal accuracy of nearly 100% from 30 epochs, while the Classic Gradient does not reach this threshold even with 200 epochs. A similar analysis can be conducted on the plot on the right, where the cost decreases much more drastically with the Stochastic Gradient than with Classic Gradient. We finally chose the value of **125 epochs**, which allows us to obtain both perfect accuracy and extremely low cost with the Stochastic Gradient, but also to optimize the two other hyperparameters.

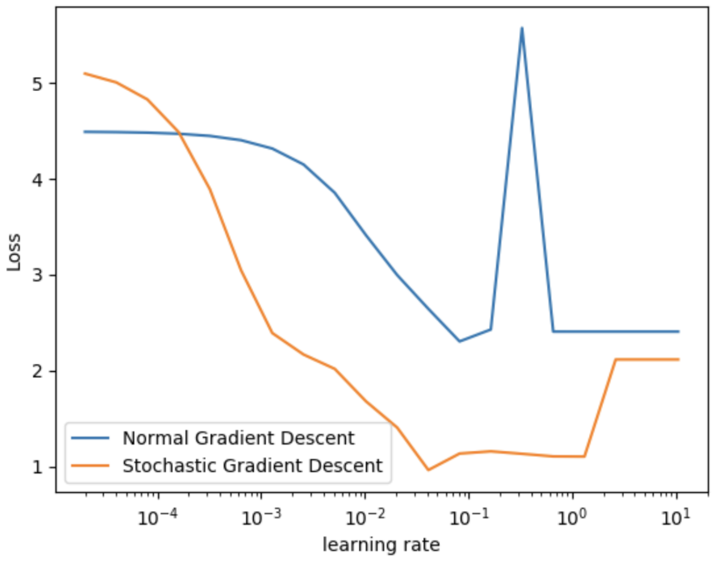


Figure 6: Cost Function against Learning Rate

As a result, we focused on finding the optimal value of the learning rate. We have plotted in Figure 6 the cost as a function of the learning rate for a Stochastic Gradient with 125 epochs. First of all, we can notice that the Stochastic Gradient Descent algorithm obtains almost systematically (again) better results than the Classic one for a given learning rate (except below $5 \cdot 10^{-3}$, too low to allow the Stochastic Gradient to converge correctly with this dataset).

Let us also notice a certain aberration for the Classical Gradient around a learning rate of $5 \cdot 10^{-1}$, which we interpreted as a situation where the algorithm gets out of the global minimum (reached with a learning rate of $8 \cdot 10^{-2}$) because of 'jumps' too large at each iteration of the algorithm (which translates to the fact that the learning rate has become too high) before falling back to a local minimum. We have, all things considered, retained for the Stochastic Gradient a value of $4 \cdot 10^{-2}$ for the **Learning Rate**.

Finally, in our refinement of the Stochastic Gradient Algorithm, we wanted to determine the optimal number of mini-batches to create to obtain a minimum cost. To do so, Figure 7 shows that the optimal value is **34 mini-batches**. Note also that we have plotted this curve up to 120 mini-batches, corresponding to the 'real' stochastic gradient (where we use a single example by iteration, and there are 120 in the training set). It turns out that this value is not the most optimal (although of very little). This confirms the idea that sometimes it might be necessary to use mini-batches of larger size than a single element to obtain great results and performances, and that the Stochastic Gradient $-$*i.e.* with only one example for the estimation of the gradient $-$ is not necessarily the most efficient as it depends (among other things) on the structure of the input data. It is also crucial to take into account the computational power needed, which decreases when the number of batches increases (since the estimation of the gradient is done much more easily and the convergence is quicker), and it then necessary to find a compromise (which wasn't exactly mandatory for us given the size of our dataset). For all these reasons, we have decided to retain the value of **120 mini-batches** as the optimal value.

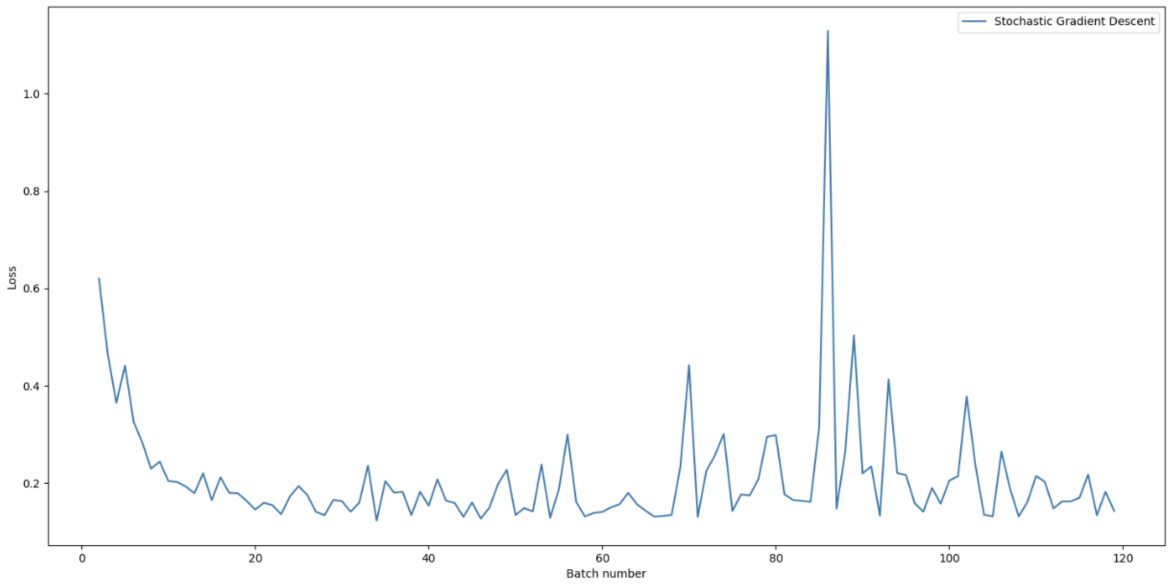


Figure 7: Cost function against Number of Batches

We have therefore proved in this section that the Stochastic Gradient Descent Algorithm was much more efficient than the Classic Gradient Descent Algorithm, especially in terms of speed of convergence (as shown in Figure 5 and Figure 6) but also in terms of the value of the minima (since it often ends up in much more interesting and $-$generally$-$ global minima). We have also determined the optimal hyperparameters of our algorithm : $\boxed{\text{Epochs} = 125, \text{ Learning Rate} = 4 \cdot 10^{-2}, \text{ Number of Batches} = 120}$. We will now focus on a method aiming at solving *Overfitting* problems.

5 Dropout Method

5.1 General Principle

Neural Networks are, as we have shown with the two previous commonly used optimization methods, extremely powerful Machine Learning tools, especially when they become very deep (with several hidden layers of neurons), allowing them to detect very complex relationships within the data [Srivastava et al., 2014]. However, the deepening of these networks is often accompanied – in addition to an extended execution time – of an almost inevitable problem: Overfitting. This is a phenomenon that exists in both Statistical and Machine Learning, during which the predictions made on the Training Set correspond more or less perfectly to the real values, but where the algorithm performs really poorly on the test set (or new data), that is to say for predictions [Wikipedia, 2018]. This is mainly due to the fact that the algorithm captures spurious correlations (*e.g.* noise) that will not be recurrent and will not represent the real underlying structure of the data and will have many difficulties generalizing what it learned to new data.

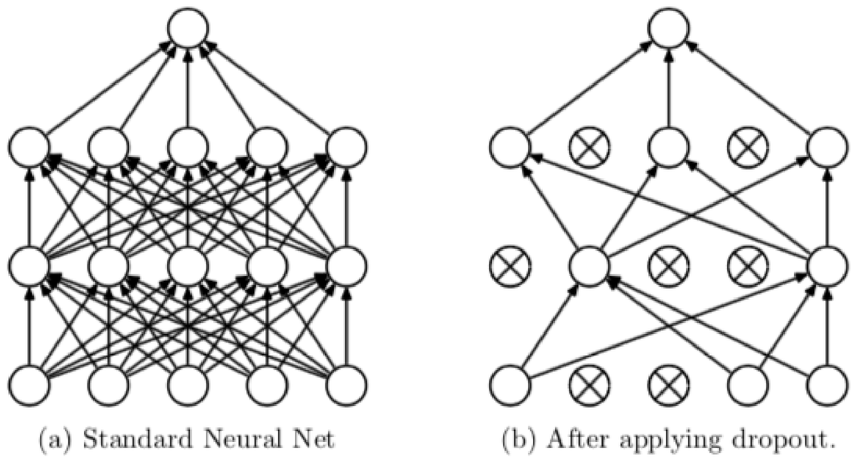


Figure 8: Dropout Method for Deep Neural Networks [Srivastava et al., 2014]. **(a)** MLP with two hidden layers. **(b)** Same MLP after application of Dropout.

The most obvious way to solve an Overfitting problem is to average the predictions of all the possible models (predictions that depend on the parameters of the said models) by weighting each one by its conditional probability knowing the data: this way of doing things is part of the larger family of methods called *Model Combination*. However, it can very quickly become very expensive and greedy for complex models like deep neural networks, to which is added the difficulty of finding optimal hyperparameters for each model. The Dropout method mainly aims at effectively overcoming these Overfitting problems in deep neural networks.

The principle of the method (illustrated in Figure 9) is quite simple, intuitive, and very inexpensive in terms of computation time: the goal is to **randomly** and **temporarily** delete (with a certain probability p) neurons from our network (with all the links associated with them, inbound as well as outbound, *cf.* Figure 8) for each learning step. Then, for the test step (on new data for example), we use the complete network by weighting each retained weight of the neurons by the probability p to counterbalance the absence of some of these neurons during the learning stages. This makes it possible to combine an exponential number of different neural networks into a single one at the test step in an extremely efficient manner.

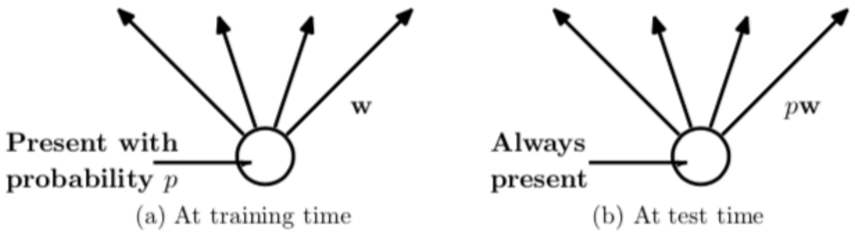


Figure 9: Illustration of Training and Test times with Dropout [Srivastava et al., 2014]. **(a)** A neuron is kept with a probability p during Learning. **(b)** Neurons are always present but their associated weights are weighted by a factor p.

5.2 Algorithm Implementation

We encountered several difficulties for the implementation of the algorithm. Indeed, the first one has been to find a way to delete (randomly using a Binomial law of parameter p) rows and columns of our weight or data matrices (which corresponds to deleting the associated nodes) both in the forward and backward propagations without getting any array size error. Right after finding a solution to this problem, another one appeared: for the test, the multiplication of all post-training weights by the probability p – as recommended in the original paper [Srivastava et al., 2014] – did not produce good results (whereas without this multiplication the results are good and even better than without the Dropout).

We then understood that we should not apply the same probability (during the training) to both input and hidden layes: we have therefore decided to find the optimal probabilities for the input and hidden layers, using a similar method to that adopted to find the hyperparameters of our Neural Network. Note that we had to increase the number of epochs to 500, otherwise there was obvious underfitting. However, this seems quite logical, because by randomly removing some neurons, the algorithm learns less quickly (but not less or with a lower quality) and it takes more passages (epochs) for the algorithm to grasp, "understand" and learn all useful correlations. It should also be noted that our study focuses on Stochastic and/or Mini-Batch Backpropagation, the advantages of which have been proved compared to the Classic Gradient Backpropagation in Section 4.3.

We heave thus plotted the two following graphs, the former bewing the one where p_{input} varies for a fixed value of $p_{hidden} = 0,15$ (Figure 10), while the latter corresponds to the one where p_{hidden} varies for a fixed value of $p_{input} = 0.1$ (Figure 11). The goal was then to successively choose the optimal values of the probabilities p_{input} and p_{hidden}. After analyzing Figure 10, we have found that for a fixed value of p_{hidden}, the optimal value of $\boxed{p_{input} \in [0.1, 0.2]}$. We have also noticed that for high values of p_{input}, the algorithm learns almost nothing (which is also know as Underfitting). Moreover, the analysis of Figure 11 led us to the conclusion that the optimal value of $\boxed{p_{hidden} \in [0.5, 0.6]}$, and that low values of p_{hidden} led to the same Underfitting problem.

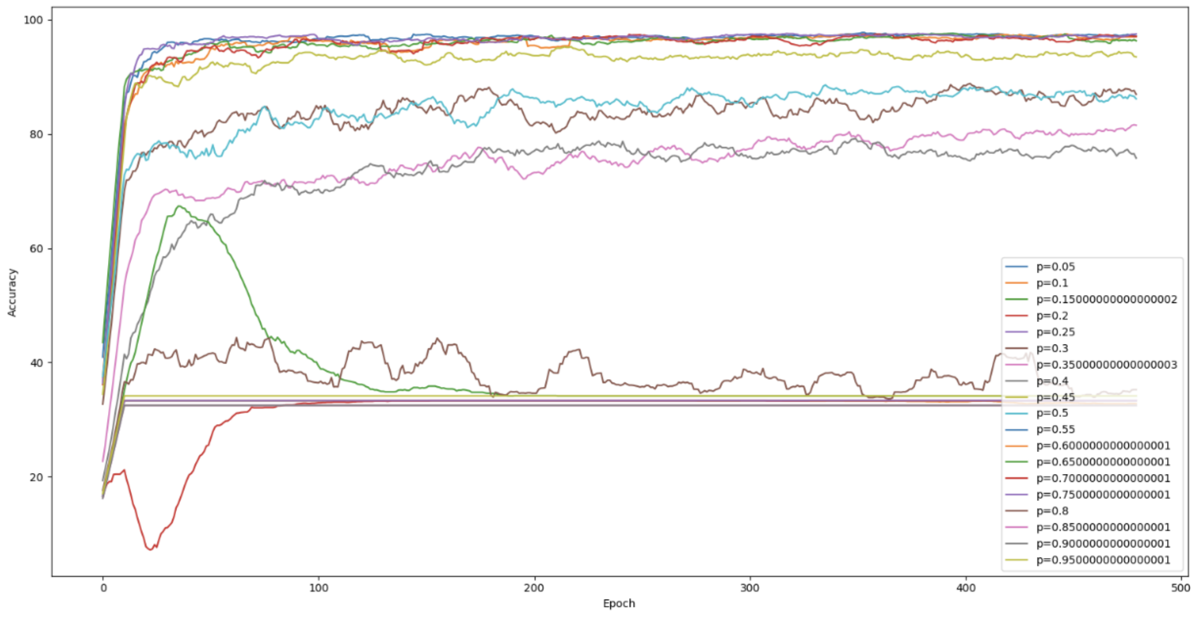


Figure 10: Accuracy against Number of Epochs with variation of p_{input}

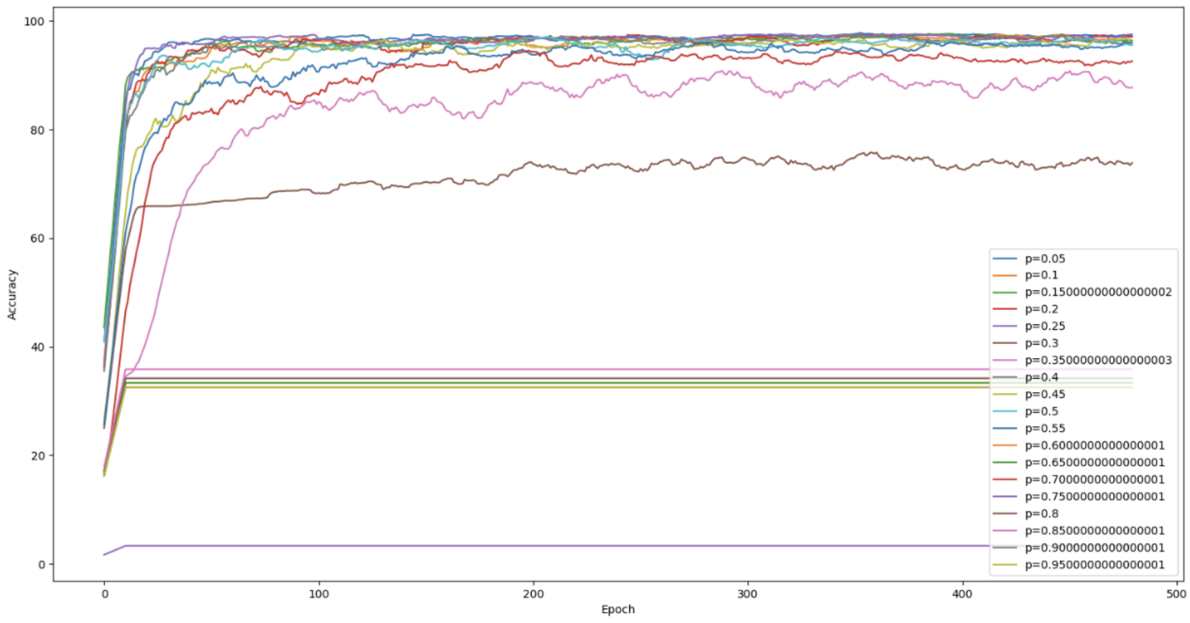


Figure 11: Accuracy against Number of Epochs with variation of p_{hidden}

However, despite this in-depth study to strictly adhere to the original paper [Srivastava et al., 2014], it seems that this multiplication by the probability p at at test time is not essential, as shown by this implementation or even this one that we found and that seem to work perfectly without it.

In conclusion, the Dropout method does not seem to be very adapted to our dataset, which is far too small for us to see any real overfitting phenomena (except when we considerably deepen our Neural Network, since it ends up learning too many things that will not be found in a test dataset, that is to say spurious correlations). In addition, the computation time has increased considerably in relation to a Stochastic Gradient without Dropout, which makes us wonder whether our implementation is the most efficient, besides comforting us in the idea that our dataset is not adapted. However, the results with the optimal probabilities remain extremely satisfactory (same accuracy of 100% despite a slower convergence), even if having access to a more consistent dataset would have been more suitable to actually witness the virtues of the Dropout method. The complete code (with Possibility of Stochastic and / or Dropout Backpropagation) is available in Appendix A - Neural Network Code.

6 Conclusion

We have thus been able, through this in-depth study, to map out, not without difficulty, the main characteristics of two optimization algorithms that are extremely important in the context of the implementation of Deep Neural Networks, the Classic and Stochastic Gradient Descent Algorithms. While the former often leads to convincing results, we have seen that the latter often makes it possible to obtain results more quickly and efficiently, while avoiding, thanks to the randomness of the algorithm, the unsatisfactory local minima of our cost function in which the classical gradient could remain blocked. We have also been able to analyze and implement the Dropout method to overcome some Overfitting problems in Deep Neural Networks in an extremely simple and efficient way.

Altogether, this project has allowed us to acquire many fundamental principles and elements of the implementation and efficient use of Deep Neural Networks, tools that have become almost inevitable in the field of Artificial Intelligence and, more precisely, Machine Learning. It has also given us the opportunity to develop our knowledge in Optimization (convex or non-convex) and Statistical Learning, thanks to the proofs of convergence that we have had the opportunity to analyze and explain with our own knowledge in mathematics. Finally, it has also given us a glimpse of a researcher's work and many things it entails: laborious bibliographic research, difficulty in moving forward and to take a step back from our work, or the necessity to develop both theoretical and practical new skills to understand certain aspects of our problem.

7 Acknowledgements

We wanted to thank Paul-Henry Cournede, on the one hand for giving us the opportunity to work on a research project so full of learning and so captivating, and on the other hand for the time and help he granted us to help us move forward with our project, while pushing us to be as independent as possible. We also wanted to thank the *Stack Overflow*, *Share*LaTeXand *ScikitLearn* communities, without which this project wouldn't have led to such great results.

References

[Bottou, 1991] Bottou, L. (1991). Stochastic Gradient Learning in Neural Networks. *Laboratoire de Recherche en Informatique, Université Paris XI.*

[Ciuperca, 2013] Ciuperca, I. (February 2013). Cours Optimisation - Cours en Master M1 SITN.

[De Sa and al, 2017] De Sa, C. and al (2017). Analyzing Stochastic Gradient Descent for some NonConvex Problems. https://users.cs.duke.edu/~rongge/stoc2017ml/cdesa_stoc_workshop.pdf.

[Duchi et al., 2011] Duchi, J. et al. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research (12).*

[Goodfellow et al., 2016] Goodfellow, I. et al. (2016). *Deep Learning.* MIT Press. http://www.deeplearningbook.org.

[Kingma et al., 2015] Kingma, D. et al. (2015). ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION.

[ScikitLearn, 2017] ScikitLearn (2017). Neural Network Models. http://scikit-learn.org/stable/modules/neural_networks_supervised.html#classification.

[Srivastava et al., 2014] Srivastava, N. et al. (2014). Dropout: A simple way to prevent Neural Networks from Overfitting. *Journal of Machine Learning Research (15).*

[Taniskidou and al, 2018] Taniskidou, D. and al (2018). UCI Machine Learning Repository. https://archive.ics.uci.edu/ml/datasets/iris.

[Tibshirani, 2015] Tibshirani, R. (Fall 2015). Convex Optimization Course (10-725/36-725) - Lecture 5. Carnegie Mellon University.

[Tibshirani et al., 2017] Tibshirani, R. et al. (August 2008 - 12th printing of 2017). *The Elements of Statistical Learning - Data Mining, Inference and Prediction.* Springer, Stanford, California.

[Wikipedia, 2018] Wikipedia (Mai 2018). Overfitting. https://en.wikipedia.org/wiki/Overfitting.

8 Appendix

8.1 Appendix A - Neural Network Code

```
# Authors: Hamza TAZI BOUARDI, Younes BELKOUCHI

#neural_net.py
import numpy as np
import activation as act
import loss
import utils

# Fix random seed for debug reasons


class Net:
    """
    Neural Net Class. Used to generate and train a model.
    """

    def __init__(self, hidden_layers=(), activation=act.ReLU, loss_function=loss.CrossEntropyLoss,
    lr=0.01, descent=None, save_metrics=False, dropout=False):
        """
        Initialisation of our neural network

        :param hidden_layers: tuple, length = number of hidden layers and
        int = number of nodes in layer Ex:(10,5)
            represent 2 hidden layers, first with 10 nodes and second with 5
        :param activation: activation function on hidden layers
        :param loss_function: loss function. if CrossEntropy, classification is used.
        :param lr: learning rate
        :param descent: if set to "stochastic", use stochastic gradient descent
        :param save_metrics: if True, save some metrics for graphs
        :param dropout: if True, enable Srivasta drop out
        """
        self.weights = None
        self.biases = None
        self.learning_rate = lr
        self.loss = loss_function

        # If cross entropy, Net is a classifier
        if self.loss == loss.CrossEntropyLoss:
            self.is_classifier = True
            self.outer_activation = act.Softmax
        else:
            self.is_classifier = False
            self.outer_activation = act.Identity

        # Set a few additionnal parameters for our net
        self.layers = list(hidden_layers)  # List of hidden layers, will be updated on first fit
        self.n_layers = len(self.layers) + 1  # Total number of layers
        self.descent = descent
        self.activation = activation

        self.first_fit = True  # If true, net has never been trained

        if save_metrics:
            self.metrics = {
                "epoch": [],
                "iter": [],
                "loss": [],
                "accuracy": [],
                "lr": [],
            }
        else:
            self.metrics = None
        # Dropout parmeters
        self.srivastava = dropout
        self.entry_proba = 0.5
        self.hidden_proba = 0.5

    def _set_input_output_layer(self, x, y):
        """
        Add layers for input and define few parameters
        :param x: features vector
        :param y: labels vector
        """
        input_shape = x.shape[1]
        try:  # Sometimes, array is in shape (n_sample,) This means that output value is only one
            output_shape = y.shape[1]
        except IndexError:
            output_shape = 1

        self.layers.insert(0, input_shape)  # add input shape

        if self.is_classifier:  # If net is classifier, define labels and classes and add last layer
            self.labels = self._get_classes(y)
            self.n_classes = len(self.labels)
            self.layers.append(self.n_classes)  # Last layer contains nodes number = to number
            of classes
        else:
            self.layers.append(output_shape)
        return

    def _get_classes(self, y):
        """
        Get all classes from labels vector
        :param y: Labels vector
        :return: dictionary of classes indexed from 1 to n_classes
        """
        labels = np.unique(y)
        return {i: labels[i] for i in range(len(labels))}

    def _fit_labels(self, y):
        """
        In case of classification, transform label vector to vector of binary values
        For example: if classes are 0,1,2, then fit_labels returns vector [1,0,0], [0,1,0], [0,0,1]
        for classes respectively 0,1,2
        :param y: labels vector
        :return: array of new labels
        """
        new_labels = [[0] * self.n_classes for i in range(len(y))]
        for i in range(len(y)):
            new_labels[i][self.labels[y[i]]] = 1
        return np.array(new_labels)

    def _reverse_labels(self, y_pred):
        """
        Reverse fit_labels method, to obtain the class of the output vectors from the neural net,
        in case of classification
        :param y_pred: vector of binary labels
        :return: array of labels vector
        """
        return np.array([self.labels[np.argmax(e)] for e in y_pred])

    def _intialize_weights(self):
        """
        Initialize weights randomly
        """
        self.biases = [np.random.randn(1, y) for y in self.layers[1:]]
        self.weights = [np.random.randn(x, y) / np.sqrt(x) for x, y in
        zip(self.layers[:-1], self.layers[1:])]

    def check_dimensions(self, inputs, outputs):
        """
        Check dimensions of inputs and outputs sothat they correspond to the layers
        :param inputs: Features Vector
        :param outputs: Labels Vector
        """
        if self.layers[0] != inputs.shape[1]:
            raise ValueError(
                "Incorrect dimension of features: Expected {} got {}".format(self.layers[0],
                inputs.shape[0]))
        try:
            if self.layers[-1] != outputs.shape[1]:
                raise ValueError(
                    "Incorrect dimension of outputs: Expected {} got {}".format(self.layers[-1],
                    outputs.shape[0]))
        except IndexError:
            if self.layers[-1] != 1:
                raise ValueError(
                    "Incorrect dimension of outputs: Expected {} got {}".format(self.layers[-1], 1))
        return

    def adjust_weights(self, nabla_weights, nabla_bias):
        """
        Adjust weight of the Net using their gradients
        :param nabla_weights: gradient of weight
        :param nabla_bias: gradient of biases
        :return:
        """
        for i in range(len(self.layers) - 1):
            self.weights[i] -= self.learning_rate * nabla_weights[i]
            for j in range(len(nabla_bias[i])):
                if self.biases[i].shape == (1,):
                    self.biases[i] -= self.learning_rate * nabla_bias[i][j][0]
                else:
                    self.biases[i] -= self.learning_rate * nabla_bias[i][j]
        return

    def feed_forward(self, input):
        """
        Execute a forward pass.
        :param input: Features vectors
        :return: Predicted Labels
        """
        pred = np.array(input)
        for i in range(self.n_layers):
            if not i == self.n_layers - 1:
                pred = self.activation.f(np.dot(pred, self.weights[i]) + self.biases[i])
            else:
                pred = self.outer_activation.f(np.dot(pred, self.weights[i]) + self.biases[i])
        return pred

    def back_propagation(self, inputs, outputs):
        """
        Execute a full back propagation algorithm
        Steps:
        - Forward pass
        - Backward pass (compute gradients)
        - Adjust weights
        If dropout is activated, additionnal step where we update prior weights
        using dropout ones is added
        :param inputs: Feature vectors
        :param outputs: Labels Vectors
        """

        # Lists contaning gradients
        bias_adjustments = []
        weight_adjustments = []

        # forward pass
        activation = inputs

        if self.srivastava:  # If dropout is activated, back up weights and biases, and
        remove input nodes and weight accordingly
            self.backup_weights = self.weights.copy()
            self.backup_biases = self.biases.copy()
            masks = []  # Masks contains arrays that dictate what colums and rows have been
            removed (0 for removed and 1 for kept)
            activation, self.weights[0], _, mask = self.dropout(0, activation, proba=self.entry_proba)
            masks.append(mask)

        activations = [activation]
        layers_nodes = []
        for i in range(self.n_layers):
            z = np.dot(activation, self.weights[i]) + self.biases[i]

            # If dropout is activaed and it is not the last layer
            if self.srivastava and not i == self.n_layers - 1:
                z, self.weights[i + 1], self.weights[i], mask = self.dropout(i + 1, z,
                proba=self.hidden_proba)
                masks.append(mask)

            layers_nodes.append(z)
            if i == self.n_layers - 1:  # last layer
                activation = self.outer_activation.f(z)
            else:
                activation = self.activation.f(z)
            activations.append(activation)

        # backward pass

        # Last layer
        delta = self.loss.delta(outputs, activations[-1])
        bias_adjustments.append(np.mean(delta, 0))
        nabla_w = np.dot(activations[-2].T, delta)
        weight_adjustments.append(nabla_w)
        # Hidden layers
        for i in range(2, len(self.layers)):
            delta = np.dot(delta, self.weights[-i + 1].T)
            * self.activation.derivative(layers_nodes[- i])
            bias_adjustments.insert(0, np.mean(delta, 0))
            nabla_w = np.dot(activations[-i - 1].T, delta)
            weight_adjustments.insert(0, nabla_w)

        # Adjust weights with gradients
        self.adjust_weights(weight_adjustments, bias_adjustments)

        # If dropout, update weights into old weights
        if self.srivastava:
            # Hidden layers
            for i in range(self.n_layers - 1):
                self.weights[i] = self.fix_dropout(self.weights[i],
                self.backup_weights[i], masks[i], masks[i + 1])
            # Last Layer
            self.weights[-1] = self.fix_dropout(self.weights[-1],
            self.backup_weights[-1], masks[-1], None)

        return

    def _fit(self, inputs, outputs, epochs=500):
        for i in range(epochs):
            self.back_propagation(inputs.copy(), outputs)
            pred = self.feed_forward(inputs)
            loss = self.loss.f(outputs, pred)
            if self.is_classifier:
                acc = np.sum(
                self._reverse_labels(outputs) == self._reverse_labels(pred)) / len(outputs) * 100
                print("Iteration: {} Loss: {} Accuracy: {}".format(i, loss, acc))
            else:
                print("Iteration: {} Loss: {}".format(i, loss))
            if self.metrics is not None:
                self.metrics["epoch"].append(i)
                if self.is_classifier:
                    self.metrics["accuracy"].append(acc)
                self.metrics["loss"].append(loss)
                self.metrics["lr"].append(self.learning_rate)
```

```
def _fit_stochastic(self, inputs, outputs, iter=2, epochs=1000):
    for t in range(epochs):
        training_set = utils.shuffle(inputs.copy(), outputs)
        x_batches = np.array_split(training_set[0], iter, axis=0)
        y_batches = np.array_split(training_set[1], iter, axis=0)
        for x, y in zip(x_batches, y_batches):
            self.back_propagation(x, y)
        pred = self.feed_forward(inputs)
        loss = self.loss.f(outputs, pred)
        if self.is_classifier:
            acc = np.sum(
            self._reverse_labels(outputs) == self._reverse_labels(pred)) / len(outputs) * 100
            print("Epoch: {} Loss: {} Accuracy: {}".format(t, loss, acc))
        else:
            print("Epoch: {} Loss: {}".format(t, loss))
        if self.metrics is not None:
            self.metrics["epoch"].append(t)
            self.metrics["iter"].append(iter)
            if self.is_classifier:
                self.metrics["accuracy"].append(acc)
            self.metrics["loss"].append(loss)
            self.metrics["lr"].append(self.learning_rate)

def fit(self, inputs, outputs, **kwargs):
    """
    Training method for the neural net.
    If stochastic is on, use stochastic fit.
    Else use normal Fit.
    :param inputs: Features vector in the form of (n_samples, n_features)
    :param outputs: Label vectors. For classification, provide normal output
    vector contaning classes directly
    :param kwargs: Additionnal arguments for fit methods
    :return: None
    """
    if self.first_fit:  # Initialize weights, update layers
        self._set_input_output_layer(inputs, outputs)
        self._intialize_weights()
        self.first_fit = False

    if self.is_classifier:  # Adjust labels so that they become binary
        labels = self._fit_labels(outputs)
    else:
        labels = outputs

   # Exectute fit method
    if self.descent == 'stochastic':
        return self._fit_stochastic(inputs, labels, **kwargs)
    elif self.descent is None:
        return self._fit(inputs, labels, **kwargs)
    else:
        raise ValueError("Incorrect descent method (only accept Stochastic or None)")

def dropout(self, layer_index, activation, proba=0.5, random_state=122):
    """
    Dropout method.
    Removes Nodes from activation, and rows or columns from weights.
    Removing nodes from ith activation corresponds to removing columns from activations[i].
    To adjust weights, we remove line i from next weights and column i from previous weights.
    :param layer_index: Activation index
    :param activation: Nodes values of current layer
    :param proba: Probablity of removal
    :param random_state: State of rng (fix)
    :return: New activation, New next weights, New previous weights, Mask used for removals
    """
    rng = np.random.RandomState(random_state)
    mask = 1 - rng.binomial(size=(activation.shape[1],), n=1,
                            p=proba)  # Array of length features of activation nodes
    dropout_act = activation[:, mask == 1]
    next_weight_new = self.weights[layer_index][mask == 1, :]
    if layer_index == 0:  # No previous weight matrix for first layer
```

```
            prev_weight_new = None
        else:
            prev_weight_new = self.weights[layer_index - 1][:, mask == 1]
        return dropout_act, next_weight_new, prev_weight_new, mask

    def fix_dropout(self, new_weights, old_weights, mask_prev, mask_next=None):
        """
        Update old weight with new dropout weights.
        New weights don't have the same format as old ones, so updating them is pretty tricky.
        :param new_weights: Weights of the Net after dropout
        :param old_weights: Weights of the Net before dropout (Real weights)
        :param mask_prev: Mask used on previous nodes
        :param mask_next: Mask used on next nodes
        :return: New weights for the real Net
        """
        if mask_next is not None:
            new_w = old_weights
            if mask_next is not None:
                conf_matrix = utils.confusion_matrix(mask_next,
                                                     mask_prev)
                # Compute confusion matrix containing indexes of weights to update
                indexes = np.argwhere(conf_matrix)
                c = 0
                for i in range(len(new_weights)):
                    for j in range(len(new_weights[i])):
                        new_w[indexes[c][0], indexes[c][1]] = new_weights[i, j]
                        c += 1
        else:
            new_w = self.fix_dropout(new_weights, old_weights,
            mask_prev, np.ones(old_weights.shape[1]))
        return new_w

    def predict(self, input):
        """
        Predict with the model (equivalent to a forward pass)
        :param input: Same format as fit method
        ;return: Predicted labels, real classes if classifier
        """
        if self.first_fit:
            raise ValueError("Train model before predicting")
        pred = self.feed_forward(input)
        if not self.is_classifier:
            return pred
        else:
            return self._reverse_labels(pred)

# activation.py
class Activation:
    """Interface regrouping methods of activation function"""

    @staticmethod
    def f(x):
        raise NotImplemented()

    @staticmethod
    def derivative(x):
        raise NotImplemented()

class Sigmoid(Activation):
    """
    Sigmoid function.
    Ref: https://en.wikipedia.org/wiki/Sigmoid_function
    """

    @staticmethod
    def f(x):
        return 1 / (1 + np.exp(-x))

    @staticmethod
    def derivative(x):
        return Sigmoid.f(x) * (1 - Sigmoid.f(x))
```

```
class Softmax(Activation):
    """
    Softmax function.
    Ref: https://en.wikipedia.org/wiki/Softmax_function
    """

    @staticmethod
    def f(x):
        shift_x = np.exp(x - x.max(axis=1)[:, np.newaxis])
        return shift_x / shift_x.sum(axis=1)[:, np.newaxis]

    @staticmethod
    def derivative(x):
        return Softmax.f(x) * (1 - Softmax.f(x))

class ReLU(Activation):
    """
    RELU activation function
    Ref: https://en.wikipedia.org/wiki/Rectifier_(neural_networks)
    """

    @staticmethod
    def f(x):
        return np.maximum(x, 0)

    @staticmethod
    def derivative(x):
        return (x > 0).astype(int)

class Softplus(Activation):
    """
    Softplus activation function
    Ref: https://en.wikipedia.org/wiki/Rectifier_(neural_networks)
    """

    @staticmethod
    def f(x):
        return np.log(1 + np.exp(x))

    @staticmethod
    def derivative(x):
        return Sigmoid.f(x)

class Tanh(Activation):
    """
    Softplus activation function
    Ref: https://en.wikipedia.org/wiki/Rectifier_(neural_networks)
    """

    @staticmethod
    def f(x):
        return np.tanh(x)

    @staticmethod
    def derivative(x):
        return 1 - np.square(np.tanh(x))

class Identity(Activation):
    @staticmethod
    def f(x):
        return x

    @staticmethod
    def derivative(x):
        return np.ones(x.shape)

# loss.py
class Loss:
    @staticmethod
    def f(y, y_pred):
        raise NotImplemented()

    @staticmethod
    def delta(y, y_pred):
        raise NotImplemented()

class CrossEntropyLoss(Loss):
    @staticmethod
    def f(y, y_pred):
        """
        Compute loss function (Cross entropy loss)
        Formula: E = -1/n * Sum_to_n(yi * log(yi) + (1-yi)*log(1-yi))
        :param pred_outputs: Predicted output via neural network
        :return: Value of loss
        """
        return -1 / len(y) * np.sum(y * np.log(y_pred + 1e-10) +
        (1 - y) * np.log((1 - y_pred) + 1e-10))

    @staticmethod
    def delta(y, y_pred):
        return 1 / len(y_pred) * (y_pred - y)

class MeanSquaredError(Loss):
    @staticmethod
    def f(y, y_pred):
        """
        Compute loss function (Mean squared error)
        Formula: E= 1/2 * (target - out)^2
        :param pred_outputs: Predicted output via neural network
        :return: value of loss
        """
        return 1 / (2 * len(y_pred)) * np.sum(np.square(y - y_pred))

    @staticmethod
    def delta(y, y_pred):
        return 1 / len(y_pred) * (y_pred - y)

# utils.py
def shuffle(x, y):
    """
    Shuffle arrays simultaneously (to keep order)
    :param x: Feature vector (n_samples, n_features)
    :param y: Labels
    :return: Feature vectors and labels shuffled
    """
    randomize = np.arange(x.shape[0])
    np.random.shuffle(randomize)
    return x[randomize, :], y[randomize, :]


def confusion_matrix(mask_next, mask_prev):
    """
    Compute confusion matrix from " arrays
    :param mask_next: Next Mask
    :param mask_prev: Previous mask
    :return: Matrix of same shape as (mask_next, mask_prev)
    containing 1 for indexes that have been kept and 0 for those removed
    """
    rows = []
    for i in mask_prev:
        col = []
        for j in mask_next:
            col.append(j * i)
        rows.append(col)
    return np.array(rows)
```