A Two-Step Rule for Backpropagation

Ahmed Boughammoura*

March 2, 2023

Abstract

We present a simplified computational rule for the backpropagation formulas for artificial neural networks. In this work, we provide a generic two-step rule for the backpropagation algorithm in matrix notation. Moreover, this rule incorporates both the forward and backward phases of the computations involved in the learning process. Specifically, this recursively computing rule permits the propagation of the changes to all synaptic weights in the network, layer by layer, efficiently. In particular, we use this rule to compute both the up and down partial derivatives of the cost function of all the connections feeding into the output layer.

Key words: Artificial neural networks; back-propagation

1 Introduction

An Artificial Neural Network (ANN) is a mathematical model which is intended to be a universal function approximator which learns from data (cf. McCulloch and Pitts, [1]). In general, an ANN consists of a number of units called artificial neurons, which are a composition of affine mappings, and non-linear (activation) mappings (applied element wise), connected by weighted connections and organized into layers, containing an input layer, one or more hidden layers, and an output layer. The neurons in an ANN can be connected in many different ways. In the simplest cases, the outputs from one layer are the inputs for the neurons in the next layer. An ANN is said to be a feedforward ANN, if outputs from one layer of neurons are the only inputs to the neurons in the following layer. In a fully connected ANN, all neurons in one layer are connected to all neurons in the previous layer [3]. An example of a fully connected feedforward network is presented in Figure 1.

In the present work we focus essentially on feed-forward artificial neural networks, with L hidden layers and a transfer (or activation) function σ, and the corresponding supervised learning problem. Let us define a simple artificial neural network as follows:

$$X_0^{\text{out}} = x, \ Y_h^{\text{out}} = W_h . X_{h-1}^{\text{out}}, \ X_h^{\text{out}} = \sigma(Y_h^{\text{out}}), \ h = 1, \cdots, L \tag{1}$$

where $x \in \mathbb{R}^n$ is the input to the network, h indexes the hidden layer and W_h is the weight matrix of the h-th hidden layer. In what follows we shall refer to the two equations of (1) as the two-step recursive forward formula. The two-step recursive forward formula is very useful in obtaining the outputs of the feed-forward deep neural networks.

A major empirical issue in the neural networks is to estimate the unknown parameters W_h with a sample of data values of targets and inputs. This estimation procedure is characterized by the recursive updating or the learning of estimated parameters. This algorithm is called the backpropagation algorithm. As reviewed by Schmidhuber [4], back-propagation was introduced and developed during the 1970's and 1980's and refined by Rumelhart et al. [2]). In addition, it is well known that the most important algorithms of artificial neural networks training is the back-propagation algorithm. From mathematical point view, back-propagation is a method to minimize errors for a loss/cost function through gradient descent. More precisely, an input data is fed to the network and forwarded through the so-called layers ; the produced output is then fed to the cost function to compute the gradient of the associated error. The computed gradient is then back-propagated through the layers to update the weights by using the well known gradient descent algorithm.

As explained in [2]), the goal of back-propagation is to compute the partial derivatives of the cost function J. In this procedure, each hidden layer h is assigned an error term δ^h. For each hidden layer, the error term δ^h is derived from error terms $\delta^k, \ k = h + 1, \cdots L$; thus the concept of error back-propagation. The output layer L is the only layer whose error term δ^L has no error dependencies, hence δ^L is the element-wise product of the partial derivatives

---

* Department of Mathematics

Higher Institute of Informatics and Mathematics, Monastir, Tunisia

Email: ahmed.boughammoura@gmail.com

of the cost function with respect to the activation X_L^{out} at the output layer L and the activation function derivative σ' with respecr to the preactivation Y_L^{out}. The δ^L is then given by the following equation

$$\delta^L = \frac{\partial J}{\partial W_L} \odot \sigma'(Y_L^{\mathrm{out}}) \tag{2}$$

For the error term δ^h, this term is derived from matrix multiplying δ^{h+1} with the weight transpose matrix $(W_{h+1})^\top$ and subsequently multiplying (element-wise) the activation function derivative σ' with respect to the preactivation Y_h^{out}. Thus, one has the following equation

$$\delta^h = (W_{h+1})^\top \delta^{h+1} \odot \sigma'(Y_h^{\mathrm{out}}), \ \ h = (L-1), \cdots, 1 \tag{3}$$

Once the layer error terms have been assigned, the partial derivative $\frac{\partial J}{\partial W_l}$ can be computed by

$$\frac{\partial J}{\partial W_h} = \delta^{h+1}(X_h^{\mathrm{out}})^\top \tag{4}$$

In particular, we deduce that the back-propagation algorithm is uniquely responsible for computing weight partial derivatives of J by using the recursive equation (3) with the initialization data given by (2). The key question to which we address ourselves in the present work is the following: how could one reformulate the back-propagation in a similar manner as in the forward pass ? Equivalently, how could one reformulate the back-propagation in two-step recursive backward formula as in (1) ?

In order to provide a first answer to this question, we shall introduce the following up and down delta's terms

$$\delta_L^{\mathrm{up}} := \frac{\partial J}{\partial X_L^{\mathrm{out}}}, \ \delta_h^{\mathrm{down}} = \delta_h^{\mathrm{up}} \odot \sigma'(Y_h^{\mathrm{out}}), \ \delta_{h-1}^{\mathrm{up}} = (W_h)^\top \delta_h^{\mathrm{down}}, \ \ h = L, \cdots, 1 \tag{5}$$

Once the δ_h^{down} term have been computed, the partial derivative $\frac{\partial J}{\partial W_h}$ can be evaluated by

$$\frac{\partial J}{\partial W_h} = \delta_h^{\mathrm{down}}(X_{h-1}^{\mathrm{out}})^\top \tag{6}$$

Now, we shall give an answer by proving in the section 3 that one has the two-step recursive backward formula given by (5).

The rest of the paper is organized as follows. Section 2 outlines some notations, setting and ANN framework. Section 3 state and proof the main mathematical result of this work. Section 4 application of this method to study some simple cases. In Section 5 conclusion, related works and mention future work directions.

2 Notations, Setting and the ANN

Let us now precise some notations. Firstly, we shall denote the space of real matrices of dimension $n \times p$ by $\mathcal{M}_{n \times p}(\mathbb{R})$, $(n, p) \in (\mathbb{N}^*)^2$, any vector $X \in \mathbb{R}^n$, is considered as columns $X = (X_1, \cdots, X_n)^T$, for any $(A, B) \in (\mathcal{M}_{n \times p}(\mathbb{R}))^2$, their component-wise product (the so-called Hadamard product) is denoted by $A \odot B$ (which is exactly the element-wise multiplication $A * B$ in Python). Secondly, for any family of transfer functions $\sigma_i : \mathbb{R} \to \mathbb{R}, \ i = 1, \cdots, n$, we shall introduce the coordinate-wise map $\sigma : \mathbb{R}^n \to \mathbb{R}^n$ by the following formula

$$\sigma(X) := (\sigma_1(X_1), \cdots, \sigma_n(X_n))^T. \tag{7}$$

This map can be considered as an "operator" Hadamard multiplication of columns $\sigma = (\sigma_1, \cdots, \sigma_n)^T$ and $X = (X_1, \cdots, X_n)^T$, i.e., $\sigma(X) = \sigma \odot X$.

Hereafter, we shall need to recall some useful multi-variable functions derivatives notations. Firstly, for any $n, m \in \mathbb{N}^*$ and any differentiable function with respect to the variable x

$$F: \ \mathbb{R} \ni x \mapsto F(x) = \left(F_{ij}(x)\right)_{\substack{1 \le i \le m \\ 1 \le j \le n}} \in \mathbb{R}^{m \times n} \tag{8}$$

we use the following notations associated to the partial derivatives of F with respect to X

$$\frac{\partial F}{\partial x} = \left(\frac{\partial F_{ij}(x)}{\partial x}\right)_{\substack{1 \le i \le m \\ 1 \le j \le n}} \tag{9}$$

Secondly, for any $n, m \in \mathbb{N}^*$ and any differentiable function with respect to the matrix variable

$$F: \ \mathbb{R}^{m \times n} \ni X = \left(X_{ij}\right)_{\substack{1 \le i \le m \\ 1 \le j \le n}} \mapsto F(X) \in \mathbb{R} \tag{10}$$

we use the so-called **denominator layout** notation (see page 15 of [7]) for the partial derivative of F with respect to the matrix X

$$\frac{\partial F}{\partial X} = \left(\frac{\partial F(X)}{\partial X_{ij}}\right)_{\substack{1 \le i \le m \\ 1 \le j \le n}} \tag{11}$$

In particular, this notation leads to the following useful formulas: for any $q \in \mathbb{N}^*$ and any matrix $W \in \mathbb{R}^{q \times m}$ we have

$$\frac{\partial(WX)}{\partial X} = W^\top, \tag{12}$$

when $X \in \mathbb{R}^n$ with $X_n = 1$ one has

$$\frac{\partial(WX)}{\partial X} = W_\sharp^\top \tag{13}$$

where $W_\sharp$ is the matrix W whose last column is removed (this formula is highly useful in practice). Moreover, for any matrix $X \in \mathbb{R}^{m \times n}$ we have

$$\frac{\partial(WX)}{\partial W} = X^\top. \tag{14}$$

Then, by the chain rule one has for any $q, n, m \in \mathbb{N}^*$ and any differentiable function with respect to the matrices variables W, X :

$$F: \ (W, X) \mapsto Z := WX \in \mathbb{R}^{q \times n} \mapsto F(Z) \in \mathbb{R}$$

$$\frac{\partial F}{\partial X} = W^\top \frac{\partial F}{\partial Z} \ \text{ and } \ \frac{\partial F}{\partial W} = \frac{\partial F}{\partial Z} X^\top \tag{15}$$

Throughout this paper, we consider layered feedforward neural networks and supervised learning tasks. We will denote such an architecture by

$$A[N_0, \cdots, N_h, \cdots, N_L] \tag{16}$$

where N_0 is the size of the input layer, N_h is the size of hidden layer h, and N_L is the size of the output layer; L is defined as the depth of the ANN , then the neural network is called as Deep Neural Network (DNN). We assume that the layers are fully connected, i.e., neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections.
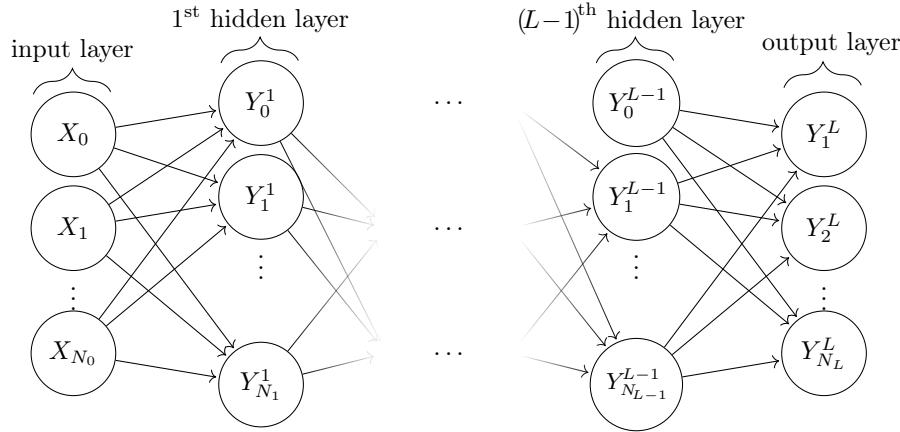
Figure 1: Example of an $A[N_0, \cdots, N_L]$ architechture.

Through the rest of the paper, we will adopt some simplified notations by replacing some subscripts and superscripts.

Now, let w_{ij}^h denote the weight connecting neuron j in layer $h - 1$ to neuron i in hidden layer h and let the associated transfer function denoted σ_i^h. In general, in the application two different passes of computation are distinguished. The first pass is referred to as the forward pass, and the second is referred to as the backward pass.

In the forward pass, the synaptic weights remain fixed throughout the network, and the output X_i^h of neuron i in hidden layer h is computed by the following recursive-coordinate form :

$$X_i^h := \sigma_i^h(Y_i^h) \quad \text{where} \quad Y_i^h := \sum_{j=1}^{N_{h-1}} w_{ij}^h X_j^{h-1} \tag{17}$$

In two-step recursive-matrix form, one may rewrite the above formulas as

$$X^h = \sigma^h(Y^h) \quad \text{where} \quad Y^h = W^h X^{h-1}, \tag{18}$$

with

$$\sigma^h := (\sigma_1^h, \cdots, \sigma_{N_h}^h)^T, \ W^h := (w_{ij}^h) \in \mathcal{M}_{N_h \times N_{h-1}}(\mathbb{R}). \tag{19}$$

Remark 1

It is crucial to remark that, if we impose the following setting on X^h, W^h and $\sigma_{N_h}^h$:

1. all input vectors have the form $X^h = [x_1, \cdots, x_{N_h-1}, 1]^T$ for all $0 \le h \le (L-1)$;

2. the last rows of all matrices W^h for all $1 \le h \le (L-1)$ have the form $[0, \cdots, 0, 1]$;

3. the last functions $\sigma_{N_h}^h$ in the columns σ^h for all $1 \le h \le (L-1)$ are identical.

Then, the $A[N_0, \cdots, N_L]$ neural network will be equivalent to a $(L-1)$-layered affine neural network with (N_0-1)-dimensional input and N_L-dimensional output. Each hidden layer h will contain (N_h-1) "genuine" neurons and one (last)"formal", responsible for the bias; the last column of the matrix W^h , except the last element, will be the bias vector for the h-th layer.

This forward pass computation between the two adjacent layers $h-1$ and h may be represented mathematically as the composition of the following two maps:

$$\mathbb{R}^{N_{h-1}} \longrightarrow \mathbb{R}^{N_h} \longrightarrow \mathbb{R}^{N_h}$$

$$X^{h-1} \overset{W^h}{\longmapsto} W^h X^{h-1} = Y^h \overset{\sigma^h}{\longmapsto} \sigma^h(Y^h) = X^h$$

It could be presented as a simple mapping diagram with X^{h-1} as input and the corresponding successive preactivation and activation $Y^h = W^h X^{h-1}, X^h = \sigma^h(Y^h)$ as outputs (see Figure 2).
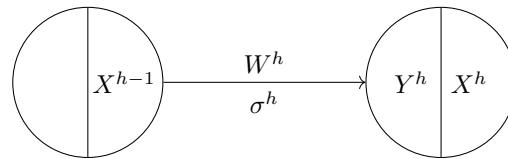
Figure 2: Mapping diagram associated the forward pass between two adjacent layers.

As consequence, the simplest neural network can be defined as a sequence of matrix multiplications and non-linearities:

$$X^0 = x, \ Y^h = W^h X^{h-1}, \ X^h = \sigma^h(Y^h), \ h = 1, 2, \cdots, L.$$

where $x \in \mathbb{R}^n$ is the input to the network, h indexes the layer and W^h is the weight matrix of the h-th layer. To optimize the neural network, we compute the partial derivatives of the cost $J(.)$ w.r.t. the weight matrices $\frac{\partial J(.)}{\partial W^h}$. This quantity can be computed by making use of the chain rule in the back-propagation algorithm. To compute the partial derivative with respect to the matrices variables $\{X^h, Y^h, W^h\}$, we put

$$\delta_h^{\text{up}} = \frac{\partial J(.)}{\partial X^h}, \ \delta_h^{\text{down}} = \frac{\partial J(.)}{\partial Y^h}, \ \delta_{W^h} = \frac{\partial J(.)}{\partial W^h}. \tag{20}$$

Now, by using the two-step rule for back-propagation, introduced in the previous section, one could rewrite the backward propagated values of the partial derivatives of J w.r.t. weight as follows :

$$\delta_L^{\text{up}} = \frac{\partial J(.)}{\partial X^L}, \ \delta_h^{\text{down}} = \delta_h^{\text{up}} \odot \sigma'(Y^h), \ \delta_{W^h} = \delta_h^{\text{down}}(X^{h-1})^\top, \ \delta_{h-1}^{\text{up}} = (W^h)^\top \delta_h^{\text{down}}, \ h = L, \cdots, 2. \tag{21}$$

Furthermore, in the practical setting mentioned in Remark 1, one should replace W^h by $W_\sharp^h$ in vertu of (13). Thus, we have for all $h \in \{(L-1), \cdots, 2\}$

$$\delta_{h-1}^{\text{up}} = (W_\sharp^h)^\top \delta_h^{\text{down}}. \tag{22}$$

The backward computation between the two adjacent layers h and $h-1$ may be represented mathematically as follows:

$$\mathbb{R}^{N_{h-1}} \times \left(\mathbb{R}^{N_h} \times \mathbb{R}^{N_{h-1}} \right) \longleftarrow \mathbb{R}^{N_h} \longleftarrow \mathbb{R}^{N_h}$$

$$\left(\delta_{h-1}^{\text{up}}, \delta_{W^h} \right) = \left(\underbrace{(W^h)^\top}_{N_{h-1} \times N_h} \underbrace{\delta_h^{\text{down}}}_{N_h \times 1}, \underbrace{\delta_h^{\text{down}}}_{N_h \times 1} \underbrace{(X^{h-1})^\top}_{1 \times N_{h-1}} \right) \xleftarrow[\ (W^h)^\top(.)\ ]{\ (.)(X^h)^\top\ } \delta_h^{\text{down}} = \delta_h^{\text{up}} \odot \sigma^{h\prime}(Y^h) \xleftarrow{\ (\odot)\sigma^{h\prime}(Y^h)\ } \delta_h^{\text{up}}$$

The simple mapping diagram below shows the two-step rule for computing the partial derivatives of the cost function w.r.t. weights (see Figure 3).
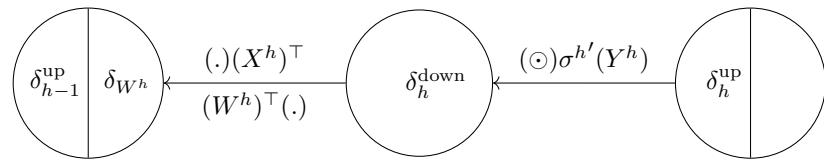
Figure 3: Mapping diagram associated the two-step backward pass between two adjacent layers.

One could combine the forward and backward passes by the following diagram, which shows clearly the two-step rule for the entire back-propagation process (see Figure 4).
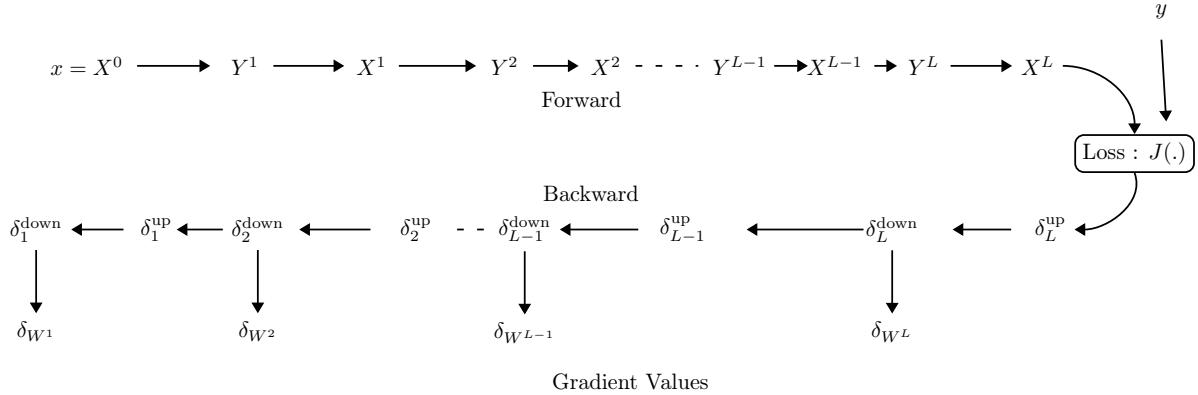
Figure 4: Mapping diagram associated the entire back-propagation process.

3   Main mathematical result

In this section we state our main result in the following Proposition. Given any differentiable cost function $J(.)$, the weight gradient of the hidden layer h is defined by all the partial derivatives of J with respect to the weights, such as:

Proposition 1 (The gradient backward propagation)

Let L be the depth of a Deep Neural Network and n_l the number of neurons in the layer l. We denote by $X^0 \in \mathbb{R}^{n_0}$ the inputs of the network, $W^h \in \mathbb{R}^{n_l} \times \mathbb{R}^{n_{h-1}}$ the weights matrix defining the synaptic strengths between the hidden layer h and its preceding $h-1$. The output Y^h of the hidden layer h are thus defined as follows:

$$\begin{cases} X^0 &= x, \\ Y^h &= W^h X^{h-1}, \\ X^h &= \sigma(Y^h), \\ h &= 1, 2, \cdots, L. \end{cases} \tag{23}$$

Where $\sigma(x)$ is a point-wise differentiable activation function. We will thus denote by $\sigma'(x)$ its first order derivative, $\sigma'(x) = \frac{\partial \sigma}{\partial x}(x)$, $X_0 = x$ is the input to the network and W^h is the weight matrix of the h-th layer. To optimize the neural network, we compute the partial derivatives of the loss $J(f(x), y)$ w.r.t. the weight matrices $\frac{\partial J(f(x),y)}{\partial W^h}$, with $f(x)$ and y are the output of the DNN and the associated target/label respectively. This quantity can be computed similarly by the following two-step rule:

$$\begin{cases} \delta_L^{\mathrm{up}} &= \dfrac{\partial J(f(x), y)}{\partial X^L}, \\ \delta_h^{\mathrm{down}} &= \delta_h^{\mathrm{up}} \odot \sigma'(Y^h) \\ \delta_{h-1}^{\mathrm{up}} &= \left(W^h\right)^T \delta_h^{\mathrm{down}}, \\ h &= L, \cdots, 1. \end{cases} \tag{24}$$

Once δ_h^{down} is computed, the weights update can be computed as

$$\frac{\partial J(f(x), y)}{\partial W^h} = \delta_h^{\mathrm{down}} \left(X^{h-1}\right)^T. \tag{25}$$

Proof of the Proposition 1

Firstly, for any $h = 1, \cdots, L$ let us recall the simplified notations introduced by (20):

$$\delta_h^{\mathrm{up}} = \frac{\partial J(f(x), y)}{\partial X^h}, \ \delta_h^{\mathrm{down}} = \frac{\partial J(f(x), y)}{\partial Y^h}.$$

Secondly, for fixed $h \in \{1, \cdots, L\}$, $X^h = \sigma(Y^h)$, then immediatly one has

$$\frac{\partial X^h}{\partial Y^h} = \sigma'(Y^h).$$

Now, by using the chain rule and (15) succsecivelly, we obtain

$$\frac{\partial J(f(x), y)}{\partial Y^h} = \frac{\partial J(f(x), y)}{\partial X^h} \cdot \frac{\partial X^h}{\partial Y^h}$$

and

$$\delta_h^{\mathrm{down}} = \delta_h^{\mathrm{up}} \odot \sigma'(Y^h). \tag{26}$$

On the other hand, $Y^h = W^h X^{h-1}$, thus

$$\frac{\partial J(f(x), y)}{\partial X^{h-1}} = (W^h)^\top \frac{\partial J(f(x), y)}{\partial Y^h}$$

in vertu of (15). As consequence,

$$\delta_{h-1}^{\mathrm{up}} = (W^h)^\top \delta_h^{\mathrm{down}}. \tag{27}$$

Equations (26) and (27) implies immediately (24). To end the proof of this Proposition, we apply again (15) to the cost function J and the relation $Y^h = W^h X^{h-1}$, we deduce that

$$\frac{\partial J(f(x), y)}{\partial W^h} = \frac{\partial J(f(x), y)}{\partial Y^h} (X^{h-1})^\top = \delta_h^{\mathrm{down}} (X^{h-1})^\top.$$

This end the proof of the Proposition 1.

4   Application to the two simplest cases of type $A[1,1,1]$

The present section shows that in the following four simplest cases associated to the DNN $A[1,1,1]$, we shall apply the two-step rule for back-propagation to compute the partial derivative of the elementary cost function J defined by $J(f(x),y) = f(x) - y$ for any real x and fixed real y. In this particular setting, we have only two simplest cases of type $A[1,1,1]$: one neuron and two neurons in the hidden layer (see Figures 5 and 6).

4.1   First case

The first simplest case corresponds to $A[1,1,1]$ architecture is shows by the Figure 5. Let us denote by a $W^1 = \begin{pmatrix} \alpha^1_{11} & \alpha^1_{12} \end{pmatrix}$ and $W^2 = \begin{pmatrix} \alpha^2_1 & \alpha^2_2 \end{pmatrix}$ the weights in the first and second layer. We will evaluate the δ_{W^1} and δ_{W^2} by the differential calculus rules firstly and then recover this result by the two-step rule for back-propagation.
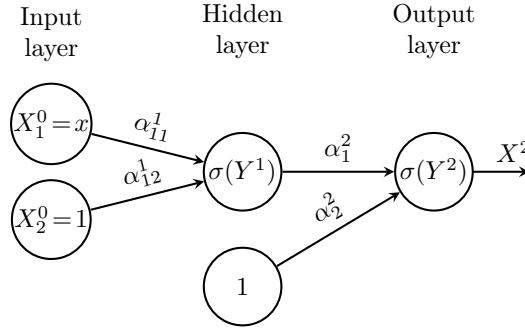
Figure 5: The DNN associated to the case 1.

The two-step forward pass :

$$X^0 = \begin{bmatrix} x \\ 1 \end{bmatrix} \xmapsto{\ \sigma^{W^1}\ } \begin{bmatrix} Y^1 = W^1 X^0 = \alpha^1_{11}x + \alpha^1_{12} \\ X^1 = \begin{pmatrix} \sigma(Y^1) \\ 1 \end{pmatrix} = \begin{pmatrix} \sigma(\alpha^1_{11}x + \alpha^1_{12}) \\ 1 \end{pmatrix} \end{bmatrix} \xmapsto{\ \sigma^{W^2}\ } \begin{bmatrix} Y^2 = W^2 X^1 = \alpha^2_1 \sigma(\alpha^1_{11}x + \alpha^1_{12}) + \alpha^2_2 \\ X^2 = \sigma(Y^2) = \sigma(\alpha^2_1 \sigma(\alpha^1_{11}x + \alpha^1_{12}) + \alpha^2_2) \end{bmatrix}$$

Hence, by using the differential calculus rules one gets

$$\delta_{W^2} = \begin{pmatrix} \sigma'(\alpha^2_1 \sigma(\alpha^1_{11}x + \alpha^1_{12}) + \alpha^2_2)\sigma(\alpha^1_{11}x + \alpha^1_{12}) \\ \sigma'(\alpha^2_1 \sigma(\alpha^1_{11}x + \alpha^1_{12}) + \alpha^2_2) \end{pmatrix}^\top$$

and

$$\delta_{W^1} = \begin{pmatrix} \sigma'(\alpha^2_1 \sigma(\alpha^1_{11}x + \alpha^1_{12}) + \alpha^2_2)\sigma'(\alpha^1_{11}x + \alpha^1_{12})\alpha^2_1 x \\ \sigma'(\alpha^2_1 \sigma(\alpha^1_{11}x + \alpha^1_{12}) + \alpha^2_2) \end{pmatrix}^\top .$$

The two-step backward pass :

$$\begin{bmatrix} \sigma'(w^1 x)\begin{pmatrix} \alpha^2_1 \sigma'(\alpha^2_1 \sigma(w^1) + \alpha^2_2) \\ \alpha^2_2 \sigma'(\alpha^2_1 \sigma(w^1) + \alpha^2_2) \end{pmatrix} = \delta_{Y^1} \\ 0 = \delta_{X^0} \end{bmatrix} \xleftarrow{\ \frac{\sigma'}{(W^1_\sharp)^\top}\ } \begin{bmatrix} \sigma'(\alpha^2_1 \sigma(w^1) + \alpha^2_2) = \delta_{X^2} \odot \sigma'(Y^2) = \delta_{Y^2} \\ \begin{pmatrix} \alpha^2_1 \sigma'(\alpha^2_1 \sigma(w^1) + \alpha^2_2) \\ \alpha^2_2 \sigma'(\alpha^2_1 \sigma(w^1) + \alpha^2_2) \end{pmatrix} = (W^2)^\top \delta_{Y^2} = \delta_{X^1} \end{bmatrix} \xleftarrow{\ \frac{\sigma'}{(W^2)^\top}\ } \delta_{X^2} = 1$$

with

$$w^1 = \alpha^1_{11}x + \alpha^1_{12}$$

Hence, by using the two-step rule one gets

$$\delta_{W^2} = \delta_{Y^2}(X^1)^\top = \begin{pmatrix} \sigma'(\alpha^2_1 \sigma(\alpha^1_{11}x + \alpha^1_{12}) + \alpha^2_2)\sigma(\alpha^1_{11}x + \alpha^1_{12}) \\ \sigma'(\alpha^2_1 \sigma(\alpha^1_{11}x + \alpha^1_{12}) + \alpha^2_2) \end{pmatrix}^\top$$

and

$$\delta_{W^1} = \delta_{Y^1}(X^0)^\top = \begin{pmatrix} \sigma'(\alpha^2_1 \sigma(\alpha^1_{11}x + \alpha^1_{12}) + \alpha^2_2)\sigma'(\alpha^1_{11}x + \alpha^1_{12})\alpha^2_1 x \\ \sigma'(\alpha^2_1 \sigma(\alpha^1_{11}x + \alpha^1_{12}) + \alpha^2_2) \end{pmatrix}^\top .$$

4.2   Second case

The second simplest case corresponds to $A[1, 1, 1]$ architecture is shows by the Figure 6. In this case we have
$W^1 = \begin{pmatrix} \alpha_{11}^1 & \alpha_{12}^1 \\ \alpha_{21}^1 & \alpha_{22}^1 \end{pmatrix}$ and $W^2 = \begin{pmatrix} \alpha_1^2 & \alpha_2^2 & \alpha_3^2 \end{pmatrix}$. Thus, one deduce immediately that
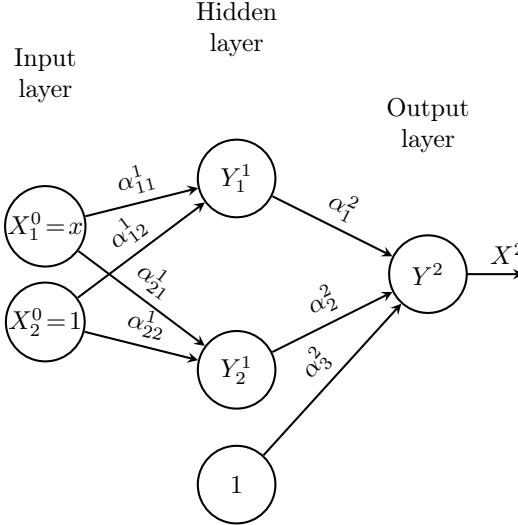
Figure 6: The DNN associated to the case 2.

The two-step forward pass :

$$X^0 = \begin{bmatrix} x \\ 1 \end{bmatrix} \xmapsto{\;\;W^1\;\;}_{\sigma} \begin{bmatrix} Y^1 = \begin{pmatrix} \alpha_{11}^1 x + \alpha_{12}^1 \\ \alpha_{21}^1 x + \alpha_{12}^1 \end{pmatrix} \\ X^1 = \begin{pmatrix} \sigma(\alpha_{11}^1 x + \alpha_{12}^1) \\ \sigma(\alpha_{21}^1 x + \alpha_{22}^1) \\ 1 \end{pmatrix} \end{bmatrix} \xmapsto{\;\;W^2\;\;}_{\sigma} \begin{bmatrix} Y^2 = \alpha_1^2 \sigma(\alpha_{11}^1 x + \alpha_{12}^1) + \alpha_2^2 \sigma(\alpha_{21}^1 x + \alpha_{22}^1) + \alpha_3^2 \\ X^2 = \sigma\Big(\alpha_1^2 \sigma(\alpha_{11}^1 x + \alpha_{12}^1) + \alpha_2^2 \sigma(\alpha_{21}^1 x + \alpha_{22}^1) + \alpha_3^2\Big) \end{bmatrix}$$

Hence, by using the differential calculus rules one gets

$$\delta_{W^2} = \begin{pmatrix} \sigma'\Big(\alpha_1^2 \sigma(w^1) + \alpha_2^2 \sigma(w^2) + \alpha_3^2\Big)\sigma(w^1) \\ \sigma'\Big(\alpha_1^2 \sigma(w^1) + \alpha_2^2 \sigma(w^2) + \alpha_3^2\Big)\sigma(w^2) \\ \sigma'\Big(\alpha_1^2 \sigma(w^1) + \alpha_2^2 \sigma(w^2) + \alpha_3^2\Big) \end{pmatrix}^\top , \ w^1 := \alpha_{11}^1 x + \alpha_{12}^1, w^2 := \alpha_{21}^1 x + \alpha_{22}^1$$

and

$$\delta_{W^1} = \begin{pmatrix} \alpha_1^2 x \sigma'(w^1) \sigma'\Big(\alpha_1^2 \sigma(w^1) + \alpha_2^2 \sigma(w^2) + \alpha_3^2\Big) & \alpha_1^2 \sigma'(w^1) \sigma'\Big(\alpha_1^2 \sigma(w^1) + \alpha_2^2 \sigma(w^2) + \alpha_3^2\Big) \\ \alpha_2^2 x \sigma'(w^1) \sigma'\Big(\alpha_1^2 \sigma(w^1) + \alpha_2^2 \sigma(w^2) + \alpha_3^2\Big) & \alpha_2^2 \sigma'(w^1) \sigma'\Big(\alpha_1^2 \sigma(w^1) + \alpha_2^2 \sigma(w^2) + \alpha_3^2\Big) \end{pmatrix} .$$

The two-step backward pass :

$$\begin{bmatrix} \begin{pmatrix} \alpha_1^2 \sigma'\Big(\alpha_1^2 \sigma(w^1) + \alpha_2^2 \sigma(w^2) + \alpha_3^2\Big)\sigma'(w^1) \\ \alpha_2^2 \sigma'\Big(\alpha_1^2 \sigma(w^1) + \alpha_2^2 \sigma(w^2) + \alpha_3^2\Big)\sigma'(w^2) \\ 0 = \delta_0^{\text{up}} \end{pmatrix} = \delta_1^{\text{down}} \end{bmatrix} \xleftarrow[(W_\sharp^1)^\top]{\sigma'} \begin{bmatrix} \sigma'\Big(\alpha_1^2 \sigma(w^1 x) + \alpha_2^2 \sigma(w^2) + \alpha_3^2\Big) = \delta_2^{\text{down}} \\ \begin{pmatrix} \alpha_1^2 \sigma'\Big(\alpha_1^2 \sigma(w^1) + \alpha_2^2 \sigma(w^2) + \alpha_3^2\Big) \\ \alpha_2^2 \sigma'\Big(\alpha_1^2 \sigma(w^1) + \alpha_2^2 \sigma(w^2) + \alpha_3^2\Big) \end{pmatrix} = \delta_1^{\text{up}} \end{bmatrix} \xleftarrow[(W^2)^\top]{\sigma'} \delta_2^{\text{up}} = 1$$

Hence, by using the two-step rule one gets

$$\delta_{W^2} = \begin{pmatrix} \sigma'\Big(\alpha_1^2\sigma(w^1) + \alpha_2^2\sigma(w^2) + \alpha_3^2\Big)\sigma(w^1) \\ \sigma'\Big(\alpha_1^2\sigma(w^1) + \alpha_2^2\sigma(w^2) + \alpha_3^2\Big)\sigma(w^2) \\ \sigma'\Big(\alpha_1^2\sigma(w^1) + \alpha_2^2\sigma(w^2) + \alpha_3^2\Big) \end{pmatrix}^{\top} \, , \ w^1 := \alpha_{11}^1 x + \alpha_{12}^1, w^2 := \alpha_{21}^1 x + \alpha_{22}^1$$

and

$$\delta_{W^1} = \begin{pmatrix} \alpha_1^2 x \sigma'(w^1)\sigma'\Big(\alpha_1^2\sigma(w^1) + \alpha_2^2\sigma(w^2) + \alpha_3^2\Big) & \alpha_1^2\sigma'(w^1)\sigma'\Big(\alpha_1^2\sigma(w^1) + \alpha_2^2\sigma(w^2) + \alpha_3^2\Big) \\ \alpha_2^2 x \sigma'(w^1)\sigma'\Big(\alpha_1^2\sigma(w^1) + \alpha_2^2\sigma(w^2) + \alpha_3^2\Big) & \alpha_2^2\sigma'(w^1)\sigma'\Big(\alpha_1^2\sigma(w^1) + \alpha_2^2\sigma(w^2) + \alpha_3^2\Big) \end{pmatrix}.$$

5   Related works and Conclusion

It is interesting to cite related works which have some connections with ours. To the best of our knowledge, in literature, the related works to this paper are [5] and [6] to the best of our knowledge. In particular, in the first paper the authors uses some decomposition of the partial derivatives of the cost function, similar to the two-step formula (5), to replace the standard back-propagation. In addition, they show (experimentally) that for specific scenarios, the two-step decomposition yield better generalization performance than the one based on the standard back-propagation. But in the second article, the authors find some similar update equation similar to the one given by (5) that report similarly to standard back-propagation at convergence. Moreover, this method discovers new variations of the back-propagation by learning new propagation rules that optimize the generalization performance after a few epochs of training.

In conclusion, we have provided a two-step rule for back-propagation similar to the one for forward propagation. We hope that it serve as a pedagogical material for data scientists, and may also inspire the exploration of novel approaches for optimizing some artificial neural networks training algorithms. As future work, we envision to explore some experimental issues to compare the performance of this two-step rule for back-propagation and the standard one.

Declarations

Conflicts of interest/Competing interests Non declared.

References

[1] McCulloch, Warren S., and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics 5 (1943): 115-133.

[2] D. E. Rumelhart, G. E. Hinton and R. J. Williams. Learning representations by backpropagating errors, Nature, vol. 323, no. 6088, pp: 533–536,1986.

[3] Gencay, Ramazan, and Tung Liu. Nonlinear modelling and prediction with feedforward and recurrent networks. Physica D: Nonlinear Phenomena 108.1-2 (1997): 119-134.

[4] Schmidhuber, Jürgen. Deep learning in neural networks: An overview. Neural networks 61 (2015): 85-117.

[5] Alber, Maximilian, et al. Backprop evolution. arXiv preprint arXiv:1808.02822 (2018).

[6] Hojabr, Reza, et al. TaxoNN: A Light-Weight Accelerator for Deep Neural Network Training. 2020 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, 2020.

[7] Ye, Jong Chul. Geometry of Deep Learning. Springer Singapore, 2022.