International Journal of Informatics and Applied Mathematics
e-ISSN:2667-6990 Vol. 6, No. 1, 57-69

A Two-Step Rule for Backpropagation

Ahmed Boughammoura

Department of Mathematics, Higher Institute of Informatics and Mathematics
Monastir, Tunisia
ahmed.boughammoura@gmail.com

Abstract. We present a simplified computational rule for the back-propagation formulas for artificial neural networks. In this work, we provide a generic two-step rule for the back-propagation algorithm in matrix notation. Moreover, this rule incorporates both the forward and backward phases of the computations involved in the learning process. Specifically, this recursive 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.

Keywords: Artificial Neural Network · Feed-Forward · Backpropagation · Delta Rule

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, [5]). 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 (cf. page 24 of [2]). 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 [7], back-propagation was introduced and developed during the 1970's and 1980's and refined by Rumelhart et al. [6]. 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 (see the diagram in Figure 5).

As explained in [6]), the goal of back-propagation is to compute the partial derivatives of the cost function J. In this procedure, for each hidden layer h is assigned an error term δ^h, then each 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 and it is given

by the following equation

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

where $\odot$ denotes the element-wise matrix multiplication (the so-called Hadamard product, which is exactly the element-wise multiplication " $*$ " in Python). For $h \in \{(L-1), \cdots, 1\}$ the error term δ^h is derived from matrix multiplication of δ^{h+1} and the weight transpose matrix $(W_{h+1})^\top$ then multiplying (element-wise) the obtained vector by the function σ' with respect to the preactivation Y_h^{out}. Thus, one has the following equation

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

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

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

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

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

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

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

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

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 [1] and [4]. 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 (cf. (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. More recently, in [3] the author have investigated the two-step rule for backpropagation from a theoretical viewpoint, and showed that the backpropagation algorithm is completely characterized by the F-adjoint of the F-propagation through the corresponding deep neural network architecture.

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.

2 Notations, Setting and the ANN

Let us now precise some notations. Firstly, we shall denote any vector $X \in \mathbb{R}^n$, is considered as columns $X = (X_1, \cdots, X_n)^\top$ and 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

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

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

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

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

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

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

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

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

we shall use the so-called **denominator layout** notation (see page 15 of [8]) for the partial derivative of F with respect to the matrix X

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

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 :

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

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

Furthermore, for any differentiable function with respect to Y

$$F:\ \mathbb{R}^n \ni Y \mapsto X := \sigma(Y) \in \mathbb{R}^n \mapsto F(X) \in \mathbb{R}$$

we have

$$\frac{\partial F}{\partial Y} = \frac{\partial F}{\partial X} \odot \sigma'(Y). \tag{16}$$

Throughout this paper, we consider layered feedforward neural networks and supervised learning tasks. Following [2] (see (2.18) in page 24), we will denote such an architecture by

$$A[N_0, \cdots, N_h, \cdots, N_L] \tag{17}$$

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.

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

Now, let α_{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 :

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

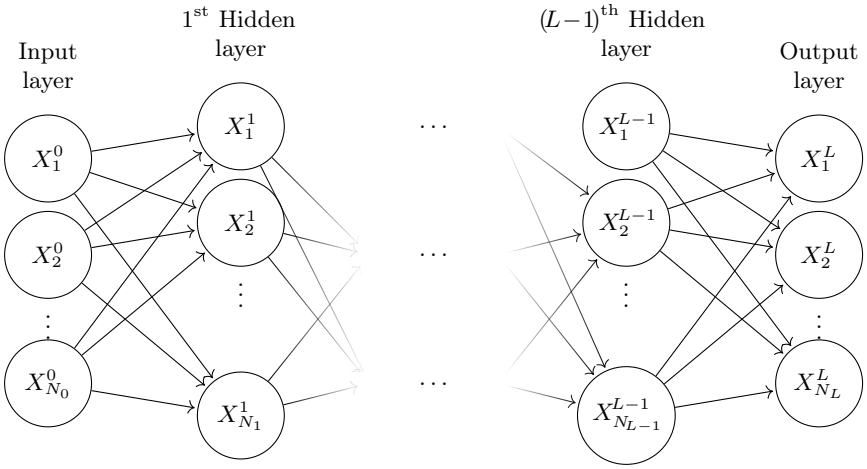

Fig. 1: Example of an $A[N_0, \cdots, N_L]$ architechture.

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

$$X^h = \sigma^h(Y^h) \text{where } Y^h = W^h X^{h-1}, \sigma^h := (\sigma_1^h, \cdots, \sigma_{N_h}^h)^\top,$$
$$W^h := (\alpha_{ij}^h) \in \mathbb{R}^{N_h \times N_{h-1}}.$$

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 $X^h = [X_1^h, \cdots, X_{N_h-1}^h, 1]^\top$ for all $0 \leq h \leq (L-1)$;
2. the last functions $\sigma_{N_h}^h$ in the columns σ^h for all $1 \leq h \leq (L-1)$ are constant functions equal to 1.

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", associated to the bias; the last column of the matrix W^h will be the bias vector for the h-th layer (For more details, see the examples given in Section 4).

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).

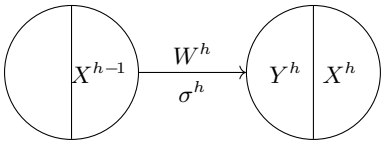

Fig. 2: Mapping diagram associated to the forward pass between two adjacent layers.

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 $x \in \mathbb{R}^{N_0}$ 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

$$\delta_h^{\text{up}} = \frac{\partial J(.)}{\partial X^h},\ \delta_h^{\text{down}} = \frac{\partial J(.)}{\partial Y^h},\ \delta_{W^h} = \frac{\partial J(.)}{\partial W^h}. \tag{18}$$

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 :

$$\begin{aligned} \delta_L^{\text{up}} = \frac{\partial J(.)}{\partial X^L},\ \delta_h^{\text{down}} = \delta_h^{\text{up}} \odot \sigma'(Y^h),\ \delta_{W^h} = \delta_h^{\text{down}} (X^{h-1})^\top, \\ \delta_{h-1}^{\text{up}} = (W^h)^\top \delta_h^{\text{down}},\ h = L, \cdots, 2. \end{aligned} \tag{19}$$

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

$$\mathbb{R}^{N_{h-1}} \times (\mathbb{R}^{N_h} \times \mathbb{R}^{N_{h-1}}) \xleftarrow{\qquad} \mathbb{R}^{N_h} \xleftarrow{\qquad} \mathbb{R}^{N_h}$$

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

$$= \delta_h^{\text{up}} \odot \sigma^{h\prime}(Y^h) \xleftarrow{(\odot)\sigma^{h\prime}(Y^h)} \delta_h^{\text{up}}$$

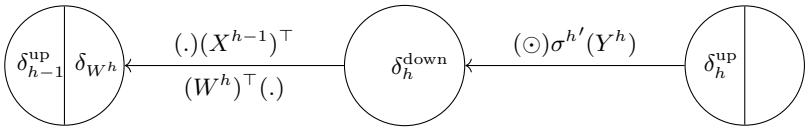

Fig. 3: Mapping diagram associated to the two-step backward pass between two adjacent layers.

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).

One may refine the above diagram to show the similarity between both the forward and backward two-step passes as follows

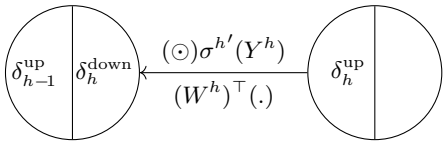

Fig. 4: Refined mapping diagram associated to the two-step backward pass between two adjacent layers.

Note that Figure 2 and Figure 4 are adjoint to each other in both computational phases. Moreover, 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 5).

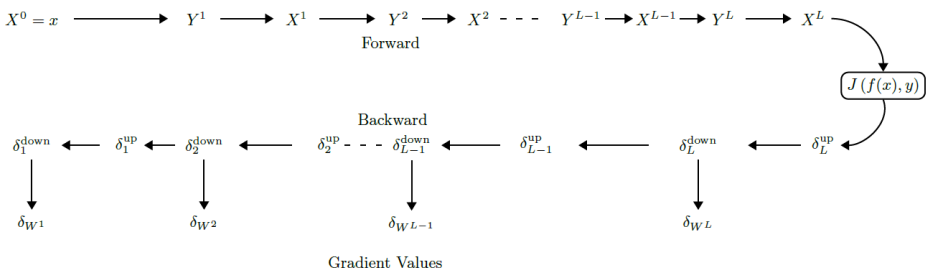

Fig. 5: Mapping diagram associated to the entire back-propagation process.

3 Main mathematical result

In this section we state our main result in the following Proposition.

Proposition 1 (The gradient backward propagation).

Let L be the depth of a Deep Neural Network and N_h the number of neurons in the h-th hidden layer. We denote by $X^0 \in \mathbb{R}^{N_0}$ the inputs of the network, $W^h \in \mathbb{R}^{N_h \times N_{h-1}}$ 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:

$$X^0 = x,\ Y^h = W^h X^{h-1},\ X^h = \sigma(Y^h),\ h = 1, 2, \cdots, L. \tag{20}$$

Where $\sigma(.)$ is a point-wise differentiable activation function. We will thus denote by $\sigma'(.)$ its first order derivative, $x \in \mathbb{R}^{N_0}$ 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:

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

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

$$\frac{\partial J(f(x), y)}{\partial W^h} = \delta_h^{\text{down}} \left(X^{h-1}\right)^\top. \tag{22}$$

Proof of the Proposition 1

Firstly, for any $h = 1, \cdots, L$ let us recall the simplified notations introduced by (18):

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

Secondly, for fixed $h \in \{1, \cdots, L\}$, $X^h = \sigma(Y^h)$, then (16) implies that

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

thus

$$\delta_h^{\text{down}} = \delta_h^{\text{up}} \odot \sigma'(Y^h). \tag{23}$$

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

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

by vertue of (15). As consequence,

$$\delta_{h-1}^{\text{up}} = (W^h)^\top \delta_h^{\text{down}}. \tag{24}$$

Equations (23) and (24) implies immediately (21). Moreover, we apply again (15) to the cost function J and the relation $Y^h = W^h X^{h-1}$, we deduce that

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

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

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

and then the associated two-step rule is given by

$$\delta_L^{\text{up}} = \frac{\partial J(f(x), y)}{\partial X^L},\ \delta_h^{\text{down}} = \delta_h^{\text{up}} \odot \sigma'(Y^h),\ \delta_{h-1}^{\text{up}} = \left(W_\sharp^h\right)^\top \delta_h^{\text{down}},\ h = L, \cdots, 1. \tag{25}$$

4 Application to the two simplest cases $A[1, 1, 1]$ and $A[1, 2, 1]$

The present section shows that in the following four simplest cases associated to the DNN $A[1, N_1, 1]$ with $N_1 = 1, 2$, 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 the two simplest cases $A[1, 1, 1]$ and $A[1, 2, 1]$: one neuron and two neurons in the hidden layer (see Figures 6 and 7).

4.1 The first case: $A[1, 1, 1]$

The first simplest case corresponds to $A[1, 1, 1]$ architecture is shows by the Figure 6. Let us denote by a $W^1 = \begin{pmatrix}\alpha_{11}^1 & \alpha_{12}^1\end{pmatrix}$ and $W^2 = \begin{pmatrix}\alpha_1^2 & \alpha_2^2\end{pmatrix}$ 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.

The two-step forward pass : For clarity, let us denote

$$y^1 := \alpha_{11}^1 x + \alpha_{12}^1 \text{ and } y^2 := \alpha_1^2 \sigma(y^1) + \alpha_2^2.$$

Obviously, one has

$$X^0 = \begin{pmatrix} x \\ 1 \end{pmatrix} \xrightarrow[\sigma]{W^1} \left\{ \begin{array}{c} Y^1 = W^1 X^0 = y^1 \\ X^1 = \begin{pmatrix} \sigma(Y^1) \\ 1 \end{pmatrix} = \begin{pmatrix} \sigma(y^1) \\ 1 \end{pmatrix} \end{array} \right\}$$

$$\xrightarrow[\sigma]{W^2} \left\{ \begin{array}{c} Y^2 = W^2 X^1 = \alpha_1^2 \sigma(y^1) + \alpha_2^2 = y^2 \\ X^2 = \sigma(Y^2) = \sigma(y^2) \end{array} \right\}$$

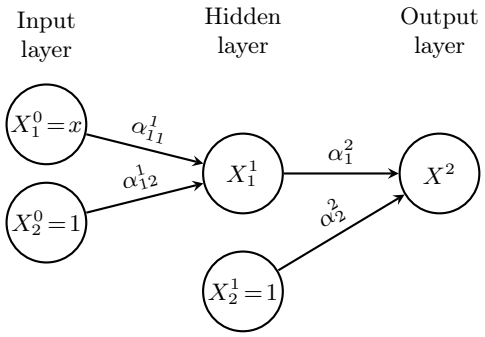

Fig. 6: The DNN associated to the case 1.

Hence, by using the differential calculus rules one gets

$$\delta_{W^2} = \begin{pmatrix} \sigma'(y^2)\sigma(y^1) \\ \sigma'(y^2) \end{pmatrix}^\top \text{ and } \delta_{W^1} = \begin{pmatrix} \sigma'(y^2)\sigma'(y^1)\alpha_1^2 x \\ \sigma'(y^2)\sigma'(y^1)\alpha_1^2 \end{pmatrix}^\top.$$

The two-step backward pass :

$$\alpha_1^2\sigma'(y^2)\sigma'(y^1) = \delta_1^{\text{down}} \xleftarrow{\sigma'} \left\{ \begin{array}{l} \sigma'(y^2) = \delta_2^{\text{up}} \odot \sigma'(Y^2) = \delta_2^{\text{down}} \\ \alpha_1^2\sigma'(y^2) = (W_\sharp^2)^\top \delta_2^{\text{down}} = \delta_1^{\text{up}} \end{array} \right\} \xleftarrow[(W_\sharp^2)^\top]{\sigma'} \delta_2^{\text{up}} = 1$$

Hence, by using the two-step rule (25) one gets

$$\delta_{W^2} = \delta_2^{\text{down}}(X^1)^\top = \begin{pmatrix} \sigma'(y^2)\sigma(y^1) \\ \sigma'(y^2) \end{pmatrix}^\top \text{ and } \delta_{W^1} = \delta_1^{\text{down}}(X^0)^\top = \begin{pmatrix} \sigma'(y^2)\sigma'(y^1)\alpha_1^2 x \\ \sigma'(y^2)\sigma'(y^1)\alpha_1^2 \end{pmatrix}^\top.$$

4.2 The second case: $A[1, 2, 1]$

The second simplest case corresponds to $A[1, 2, 1]$ architecture is shows by the Figure 7. 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

The two-step forward pass : As in the previous case, we shall denote

$$y_{11}^1 := \alpha_{11}^1 x + \alpha_{12}^1,\ y_{21}^1 := \alpha_{21}^1 x + \alpha_{22}^1 \text{ and } y^2 := \alpha_1^2\sigma(y_{11}^1) + \alpha_2^2\sigma(y_{21}^1) + \alpha_3^2.$$

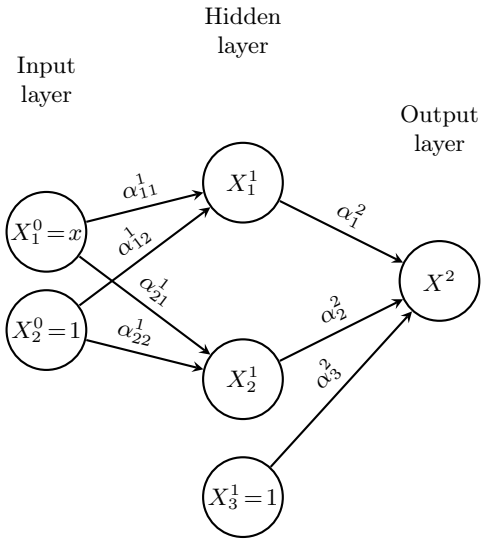

Fig. 7: The DNN associated to the case 2.

Obviously, one has,

$$X^0 = \begin{pmatrix} x \\ 1 \end{pmatrix} \overset{\substack{W^1 \\ \sigma}}{\longmapsto} \left\{ \begin{array}{c} Y^1 = \begin{pmatrix} y_{11}^1 \\ y_{21}^1 \end{pmatrix} \\ \\ X^1 = \begin{pmatrix} \sigma(y_{11}^1) \\ \sigma(y_{21}^1) \\ 1 \end{pmatrix} \end{array} \right\} \overset{\substack{W^2 \\ \sigma}}{\longmapsto} \left\{ \begin{array}{c} Y^2 = y^2 \\ X^2 = \sigma(y^2) \end{array} \right\}$$

Hence, by using the differential calculus rules one gets

$$\delta_{W^2} = \begin{pmatrix} \sigma'(y^2)\sigma(y_{11}^1) \\ \sigma'(y^2)\sigma(y_{21}^1) \\ \sigma'(y^2) \end{pmatrix}^\top$$

and

$$\delta_{W^1} = \begin{pmatrix} \alpha_1^2 x\sigma'(y_{11}^1)\sigma'(y^2) & \alpha_1^2\sigma'(y_{11}^1)\sigma'(y^2) \\ \alpha_2^2 x\sigma'(y_{21}^1)\sigma'(y^2) & \alpha_2^2\sigma'(y_{21}^1)\sigma'(y^2) \end{pmatrix}.$$

The two-step backward pass :

$$\begin{pmatrix} \alpha_1^2\sigma'(y^2)\sigma'(y_{11}^1) \\ \alpha_2^2\sigma'(y^2)\sigma'(y_{21}^1) \end{pmatrix} = \delta_1^{\text{down}} \overset{\sigma'}{\longmapsfrom} \left\{ \begin{array}{c} \sigma'(y^2) = \delta_2^{\text{down}} \\ \\ \begin{pmatrix} \alpha_1^2\sigma'(y^2) \\ \alpha_2^2\sigma'(y^2) \end{pmatrix} = \delta_1^{\text{up}} \end{array} \right\} \overset{\substack{\sigma' \\ (W_\sharp^2)^\top}}{\longmapsfrom} \delta_2^{\text{up}} = 1$$

Hence, by using the two-step rule (25) one gets

$$\delta_{W^2} = \delta_2^{\text{down}}(X^1)^\top = \begin{pmatrix} \sigma'(y^2)\sigma(y_{11}^1) \\ \sigma'(y^2)\sigma(y_{21}^1) \\ \sigma'(y^2) \end{pmatrix}^\top$$

and

$$\delta_{W^1} = \delta_1^{\text{down}}(X^0)^\top = \begin{pmatrix} \alpha_1^2 x\sigma'(y_{11}^1)\sigma'(y^2)\ \alpha_1^2\sigma'(y_{11}^1)\sigma'(y^2) \\ \alpha_2^2 x\sigma'(y_{21}^1)\sigma'(y^2)\ \alpha_2^2\sigma'(y_{21}^1)\sigma'(y^2) \end{pmatrix}.$$

5 Conclusion

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.

Acknowledgement

The author would like to thank two anonymous reviewers for pointing out several useful comments and suggestions that helped improve this manuscript.

Conflicts of Interest

The author declare no conflict of interest.

References

1. Alber, M., Bello, I., Zoph, B., Kindermans, P. J., Ramachandran, P., & Le, Q. (2018). Backprop evolution. Preprint at https://arxiv.org/abs/1808.02822.
2. Baldi, P. (2021). Deep learning in science. Cambridge University Press.
3. Boughammoura, A. (2023). Backpropagation and F-adjoint. Preprint at https://arxiv.org/abs/2304.13820.
4. Hojabr, R., Givaki, K., Pourahmadi, K., Nooralinejad, P., Khonsari, A., Rahmati, D., & Najafi, M. H. (2020, October). TaxoNN: A Light-Weight Accelerator for Deep Neural Network Training. In 2020 IEEE International Symposium on Circuits and Systems (ISCAS) (pp. 1-5). IEEE.
5. McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5, 115-133.
6. Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. nature, 323 (6088), 533-536.
7. Schmidhuber, J. (2015). Deep learning in neural networks: An overview. Neural networks, 61, 85-117.
8. Ye, J. C. (2022). Geometry of Deep Learning. Springer Singapore.