Backpropagation primer with numerical example and Python script

Nicola Santi

July 20, 2025

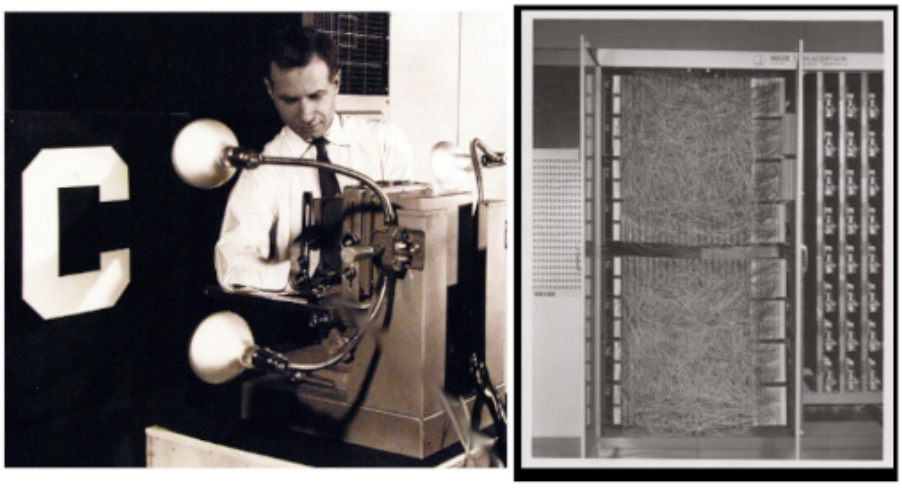
Abstract

A self-contained introduction to the well-known backpropagation algorithm illustrated step by step, providing the mathematical elements necessary for understanding and a numerical example with which to verify what has been learnt. A Python script with keras and tensorflow to verify the calculations performed completes the exposition.

1 Introduction

Backpropagation, short for 'backward propagation of errors,' is perhaps the most iconic algorithm of modern machine learning. Suffice it to say that until the mid-1980s, nobody believed in the possibility of training a multilayer network anymore! The story is very interesting and reveals how a mistaken belief can delay scientific development even by decades.

While the first simple artificial neural network dates back to Mcculloch and Pitts (1943), it was in Rosenblatt (1962) that the theory and hardware needed to implement the first neural network called perceptron was developed. The hardware needed to implement it is depicted in the photos below:



Perceptron has several layers but only one trainable so, in modern terms, it is a one-layer neural network, obviously subject to major limitations formally demonstrated in Minsky and Papert (1969). And here we come to the exact moment when machine learning based on neural networks was in danger of being completely abandoned because the two authors firmly believed, although without providing proof, that the same limitations would also apply to neural networks with more than one layer (deep learning).

The widespread belief led to a suspension of interest, funding and thus research for the period from 1970 to the first half of the 1980s. Among other things, researchers could not pursue the study of

multilayer models because there was no way to train them: the techniques used by perceptron were not applicable except for networks with only one layer.

Then, as sometimes happens, a series of innovations contributed to the revival of neural networks and, among these, the most relevant was the invention [1] of the backpropagation algorithm by Rumelhart, Hinton, and Williams (1986). The activation and loss functions had already been made differential, and backpropagation allows the derivative (more precisely the gradient) of the loss function to be calculated by moving backwards from the errors (difference between the value predicted by the model and the actual value) and going up from the last layer to the first. The most relevant aspect, as we shall see in this article, is the possibility of reusing the calculations already made by the neural network in its way forward (from the first layer to the last) to calculate the gradient, which saves resources and makes it possible to handle also even very deep neural networks.

We will begin our exposition by recalling the few elements of mathematics necessary for understanding, which are limited to the multiplication of matrices and vectors and the calculation of derivatives of linear functions. We will then formally define a multilayer neural network and focus on a numerical example with two layers. We will use what we have learned to obtain two simple and important derivatives to be used in the last paragraph to obtain the complete backpropagation algorithm. We will test our understanding by carrying out elementary calculations on our example model, and we will also write a small Python script to verify the correctness of our accounts.

Let us proceed, then `:-)`

2 Matrix multiplication (and little else)

In this article, we will use the original representation of matrix multiplication $WZ = A$ you can see below:

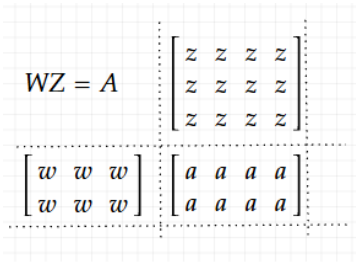


Figure 1: Two matrix multiplication

The advantage is the immediate visualisation of the dimensions of the matrix obtained, which will have as many rows as W and as many columns as Z. Remember also that the number of columns of W must coincide with the number of rows of Z for multiplication to occur.

Let us now look at the special case of multiplication between matrix and vector $W\mathbf{z} = \mathbf{a}$:

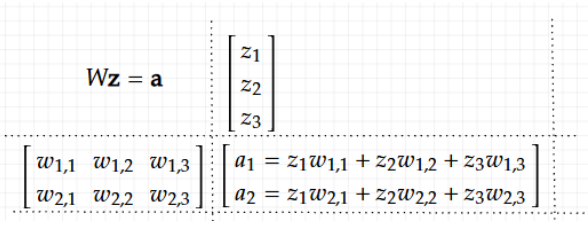


Figure 2: Matrix and vector multiplication

[1] Mathematical principles behind it date back to Kelley (1960) and one of the earliest known implementations of backpropagation can be traced back to Werbos (1974), an aerospace engineer who introduced the concept in his 1974 Ph.D. thesis at Harvard University titled "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences." Werbos recognized that this algorithm could be applied not only to neural networks but to a broader range of predictive models. Despite this early insight, his work remained largely unnoticed by the mainstream AI community for nearly a decade.

It is important to note how each element of $\mathbf{z}$ contributes to determining each element of $\mathbf{a}$: this awareness will come in handy in one of the last paragraphs of this paper (5.2). Moreover, this is precisely why layers that contain a matrix multiplication are called dense, since each input element influences each output element.

Finally, each a_k being the sum of products between elements of $\mathbf{z}$ and W, the derivative of a_k with respect to z_m is simply $w_{k,m}$ while, symmetrically, the derivative with respect to $w_{k,m}$ is z_m (see 4.1).

3 Multilayer dense network: formalization

A dense feedforward neural network can be formalized as a sequence of L layers, each of which receives an input vector $\mathbf{z} \in \mathbb{R}^M$ to which apply first a linear transformation W and, then, a nonlinear transformation $h()$. The result is used by the next layer as input (hence the feed-forward meaning), see Figure below.

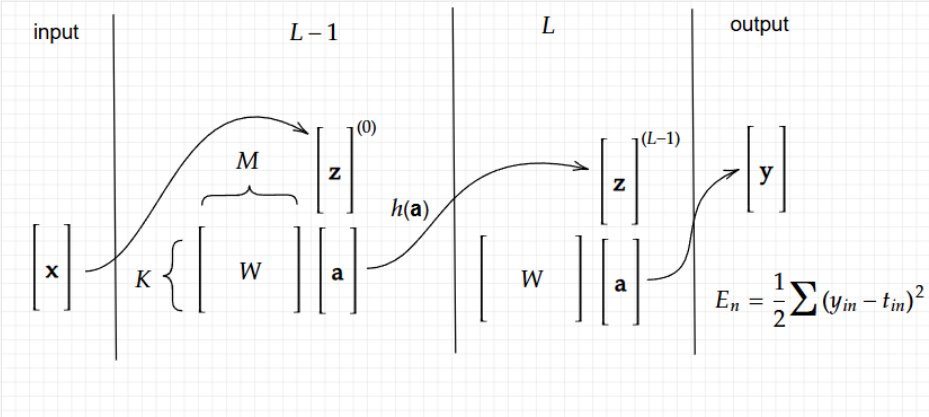


Figure 3: Simple feed-forward network

The linear transformation inside each layer multiplies the input vector $\mathbf{z}$ by the model parameter $K \times M$ matrix W (each layer has its own matrix with different dimensions M and K) so that each input neuron (each component of vector $\mathbf{z}$) is connected to each output neuron (component of vector $\mathbf{a}$). The result is the vector $\mathbf{a} \in \mathbb{R}^K$.

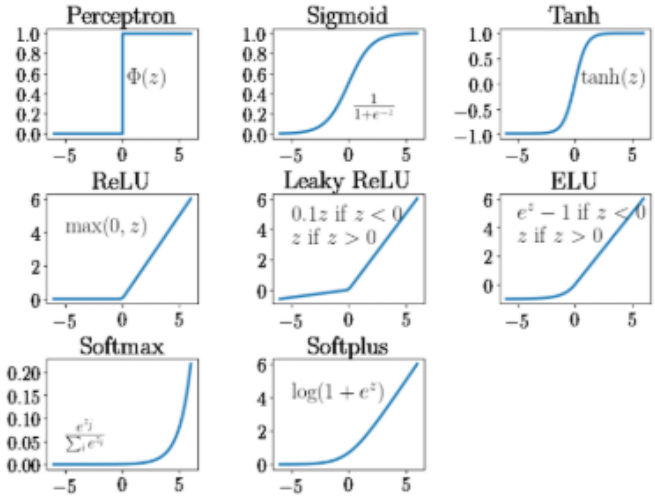


Figure 4: Some common activation functions

A potentially non-linear transformation called activation function then is applied to vector $\mathbf{a}$. Common examples of this kind of functions are sofmax, ReLU or Sigmoid (see Figure above from

www.researchgate.net). In turn, the result of this activation function becomes the input of the next layer $\mathbf{z}^{(l)} = h(\mathbf{a}^{(l)})$. The superscripts in parentheses indicate the layer to which each variable refers.

Summarizing and formalizing, for each layer l we have:

$$\begin{aligned}\mathbf{a}^{(l)} &= W^{(l)}\mathbf{z}^{(l-1)} \\ \mathbf{z}^{(l)} &= h^{(l)}\left(\mathbf{a}^{(l)}\right) \\ M^{(l)} &= K^{(l-1)}\end{aligned}$$

The loss function is simply the mean square of the errors $E_n(\mathbf{y}) = \frac{1}{2}\sum(y_{mn} - t_{mn})^2$, where n denotes a single data point.

A couple of definitions and a few remarks conclude the presentation of our network:

- $\mathbf{x} = \mathbf{z}^{(0)}$ is the input of the whole neural network;
- $\mathbf{y} = \mathbf{z}^{(L)}$ is the output of the whole neural network;
- L is the total number of layers;
- $M^{(l)}$ is the dimension of layer l input $\mathbf{z}^{(l-1)} \in \mathbb{R}^{M^{(l)}}$;
- $K^{(l)}$ is the dimension of the layer l output $\mathbf{z}^{(l)} \in \mathbb{R}^{K^{(l)}}$;
- $K^{(l-1)} = M^{(l)}$ obviously, the size of the output of each level must match the next layer input;

> **Note**
>
> For didactic purposes, the last layer L has no activation function or, and the same thing, uses the identity function $h(\mathbf{a}) = \mathbf{a}$. We will return to this point in the section 5.1. For similar reasons, each level has no bias.

The superscripts in parentheses, as said, indicate the layer to which each variable refers: they are somehow inconvenient, so, for simplicity, in case of omission it will be understood that we are referring to the current level l. So we can simplify our equations:

$$\begin{aligned}\mathbf{a} &= W\mathbf{z}^{(l-1)} \\ \mathbf{z} &= h(\mathbf{a}) \\ M &= K^{(l-1)}\end{aligned}$$

Python code: we can declare the model just described with a few lines of Python using the well-known keras library.

```
model = keras.Sequential([
    layers.Input(shape=(2,)),
    layers.Dense(3, activation='tanh',   use_bias=False),
    layers.Dense(2, activation='linear', use_bias=False)
])
```

The source code is available at github.com/nicolinux72/backpropagation.git.

4 Two simple, useful derivatives

In this section, we will try to understand how $\mathbf{a}$ varies with respect to the variables of the same layer (W and $\mathbf{z}$) and, subsequently, with respect to the tensor $\mathbf{a}^{l-1}$ of the previous layer.

We will discover that we are dealing with very simple derivatives, given the extreme linearity of the model. I recommend following the calculations in the figure below.

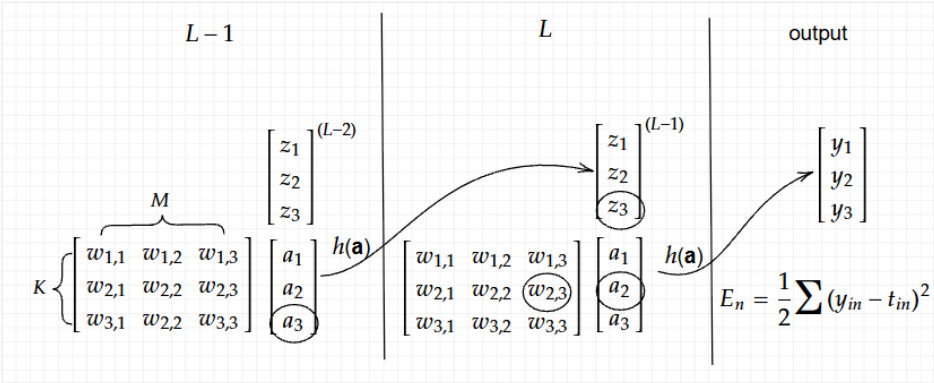


Figure 5: Derivatives of **a** with respect to the current layer L and the previous one

4.1 From same layer

In each layer l we have two simple derivatives to deal with:

$$\frac{\partial a_k}{\partial w_{k,m}} = z_m^{(l-1)} \tag{1}$$

$$\frac{\partial a_k}{\partial z_m^{(l-1)}} = w_{k,m} \tag{2}$$

Taking as a reference the circled elements in Figure 5 : we know from paragraph 2 that $a_2 = w_{2,1}z_1 + w_{2,2}z_2 + w_{2,3}z_3$ so derivative of a_2 from $w_{2,3}$ is simply z_2. Or, conversely, the derivative of a_2 from z_2 is $w_{2,3}$.

Note: the (2), on a closer inspection, already connects to the previous layer being the input of layer l the output of the previous one but it is not yet the derivative with respect to $\mathbf{a}^{l-1}$ that we will calculate in the next paragraph.

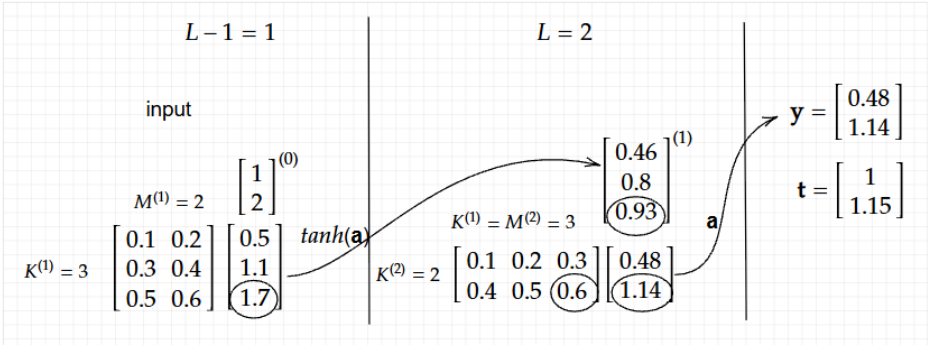


Figure 6: A numerical example

Our example: to refine the understanding, let us use the small two-layer neural network drawn above. The weights displayed are those at a certain epoch in training, the label **t** is the truth for a certain data point. All values are approximated to two decimal places:

$$\frac{\partial a_2}{\partial w_{2,3}} = z_3^{(1)} = 0.93$$

$$\frac{\partial a_2}{\partial z_3^{(1)}} = w_{2,3} = 0.6$$

Python code: possiamo ricreare il nostro esempio inizializzando i pesi di ogni livello, istanziando il tensore di x di input e l'etichetta t.

```
# set weights as in the article
w1 = np.array([[0.1, 0.2],
               [0.3, 0.4],
               [0.5, 0.6]])

w2 = np.array([[0.1, 0.2, 0.3],
               [0.4, 0.5, 0.6]])

model.layers[0].set_weights([w1.T])
model.layers[1].set_weights([w2.T])

# set also a fake data point and label
x = tf.constant([[1.0, 2.0]])
y_true = tf.constant([[1.0, 1.15]])
```

4.2 From previous layer

As noticed, the derivative (2) already links current layer to previous one, because $\mathbf{z}^{(l-1)}$ is both previous layer output and current layer input. But we need do be linked to previous $\mathbf{a}^{l-1}$. It's an easy task remembering that $z_m^{(l-1)} = h^{(l-1)}(a_m^{(l-1)})$, so:

$$\frac{\partial z_m^{(l-1)}}{\partial a_m^{(l-1)}} = \dot{h}^{(l-1)}(a_m^{(l-1)}) \tag{3}$$

This, obviously, involves the derivative of activation function $h^{l-1}()$.

Our example: Again, using Figure (6), we calculate the derivative of the third component of the first-level output $z_3^{(1)}$ (which is also the third component of the second-level input) based on the $a_3^{(1)}$. Remember that, in our example, the activation of layer $L-1$ is $tanh()$:

$$\frac{\partial z_3^{(1)}}{\partial a_3^{(1)}} = \dot{tanh}(1.7) = 1 - tanh^2(1.7) = 1 - (0.93)^2$$

Then, using the (2) and the 3, it is possible to obtain the derivative of $\mathbf{a}$ with respect to the previous $\mathbf{a}^{(l-1)}$:

$$\frac{\partial a_k^{(l)}}{\partial a_m^{(l-1)}} = \underbrace{\frac{\partial z_m^{(l-1)}}{\partial a_m^{(l-1)}}}_{(3)} \underbrace{\frac{\partial a_k^{(l)}}{\partial z_m^{(l-1)}}}_{(2)} = \dot{h}^{(l-1)}(a_m^{(l-1)})\ w_{k,m}^{(l)} \tag{4}$$

Our example: we simply apply the (4)

$$\frac{\partial a_2^{(2)}}{\partial a_3^{(1)}} = [(1 - (0.93)^2]0.6$$

5 Backpropagation algotrithm

To train our model we need to compute the derivative (more precisely the gradient) of each loss function $E_n(\mathbf{y}) = \frac{1}{2}\sum(y_{in} - t_{in})^2$ with respect to all model parameters $w_{k,m}$.

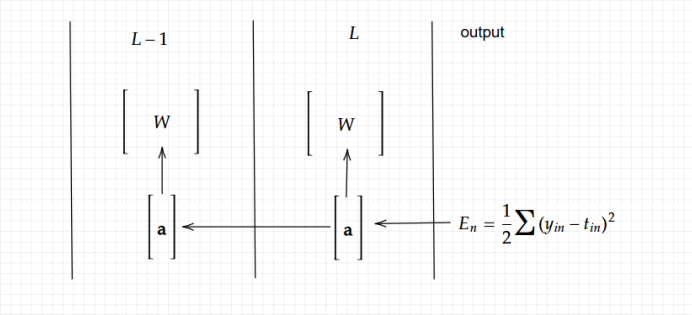


Figure 7: Backpropagation move from layer to previuos one using $\mathbf{a}$ derivatives and then compute gradient respect with W.

The basic idea of backpropagation is to calculate the derivative of E with respect to all $\mathbf{a}$ starting from the last layer L and proceeding backward using (4) , moving in reverse we could reuse the calculations already made during the forward way.

Having obtained the derivatives with respect to each $\mathbf{a}$, we have all the information needed to calculate, using (5.1), the derivatives with respect to each W which are the final objective of our algorithm.

Let us formalize backpropagation algorithm in the easy steps below:

1. Derive E with respect to last layer $\mathbf{a}^L$;
2. Calculate the derivative of $\mathbf{a}^L$ with respect to W^L using (5.1);
3. Using (4) we move to previous layer $L-1$ and derive $\mathbf{a}^L$ with respect to $\mathbf{a}^{L-1}$
4. Repeat steps 2 and 3 until we reach the first layer

Let's do it!

5.1 Last layer

We begin by noting how the derivative of $E(W)$ with respect to the last layer has an understandable meaning (remembering that $\mathbf{y} = \mathbf{a}^{(L))}$):

$$\frac{\partial E}{\partial a_k^{(L)}} = \frac{\partial \frac{1}{2}\sum(a_i - t_i)^2}{\partial a_k^{(L)}} = \frac{\partial \frac{1}{2}\sum(y_i - t_i)^2}{\partial a_k^{(L)}} = \; y_k - t_k$$

It is exactly error of the component k so, for convenience of notation, we define $\delta_k^{(l)}$ and call it *error*:

$$\delta_k^{(l)} \equiv \frac{\partial E}{\partial a_k^{(l)}} \tag{5}$$

Please note:

As said, in our simplified model the last layer has no activation function: if there were, it would have to be taken into account in exactly the same way as (3) but the interpretation of δ as closely related to the error measure will remains valid.

Our example: We apply step 1 to calculate the error δ^L of our example described in Figure 6:

$$\delta_1^{(2)} = \frac{\partial E}{\partial a_1^{(2)}} \;=\; (0.48 - 1) \;= -0.52$$

$$\delta_2^{(2)} = \frac{\partial E}{\partial a_2^{(2)}} \;=\; (1.14 - 1.15) \;= -0.01$$

We complete the last layer by deriving $\mathbf{a}$ with respect to each element W. We are helped by (5.1) and (5).

$$\frac{\partial E}{\partial w_{k,m}^L} = \frac{\partial E}{\partial a_k^L}\frac{\partial a_k^L}{\partial w_{k,m}} \;=\; \delta_k^L \; z_m^{(l-1)}$$

Our example: compute derivative for each $w_{k,m}$ of last layer L

$$\frac{\partial E}{\partial w_{1,1}^{(2)}} \;=\; -0.52\ (0.46) = -0.24\ ; \quad \frac{\partial E}{\partial w_{1,2}^{(2)}} \;=\; -0.52\ (0.80) = -0.41\ ; \quad \frac{\partial E}{\partial w_{1,3}^{(2)}} \;=\; -0.52\ (0.93) = -0.48$$

$$\frac{\partial E}{\partial w_{2,1}^{(2)}} \;=\; -0.01\ (0.46) = -0.00\ ; \quad \frac{\partial E}{\partial w_{2,2}^{(2)}} \;=\; -0.01\ (0.80) = -0.01\ ; \quad \frac{\partial E}{\partial w_{2,3}^{(2)}} \;=\; -0.01\ (0.93) = -0.01$$

5.2 Previous layer

Returning to the level $L-1$, we should note how the modification of each $a_m^{(L-1)}$ impacts on each $a_k^{(L)}$ as anticipated in Paragraph 2. With the help of Figure 6, let us convince ourselves that each element k of $\mathbf{a}^L$ is in fact modified by $a_m^{(L-1)}$ because it equals $a_k = ... + \ w_{k,m} z_m + ...$ In other words, each row (vector) of the matrix W is multiplied by $\mathbf{z}^{L-1}$. Please note how the variation is weighted by the coefficient $w_{k,m}$.

Thus, using the usual rules of derivation, we obtain that:

$$\delta_m^{(L-1)} = \frac{\partial E}{\partial a_m^{(L-1)}} \;= \sum_k^{K^{(L)}} \underbrace{\frac{\partial z_m^{(L-1)}}{\partial a_m^{(L-1)}}\frac{\partial a_k^{(L)}}{\partial z_m^{(L-1)}}}_{(4)} \underbrace{\frac{\partial E}{\partial a_k^{(L)}}}_{(5)}$$

$$= \dot{h}^{(L-1)}\left(a_m^{(L-1)}\right) \sum_k^{K^{(L)}} \; w_{k,m}^{(L)} \; \delta_k^{(L)}$$

We have used the (4) and (5) where indicated.

Our example: I think it may be useful to train intuition with the simple calculations for the penultimate $L-1$ layer (which, in our example, is also the first). Starting with $\boldsymbol{\delta^{L-1}}$

$$\begin{aligned}
\delta_1^{(1)} &= tan\dot{h}(0.5)[\ 0.1\ (-0.52) + \ 0.4(-0.01)] = -0.044 \\
\delta_2^{(1)} &= tan\dot{h}(1.1)[\ 0.2\ (-0.52) + \ 0.5(-0.01)] = -0.039 \\
\delta_3^{(1)} &= tan\dot{h}(1.7)[\ 0.3\ (-0.52) + \ 0.6(-0.01)] = -0.020
\end{aligned}$$

Now compute the derivation with respect to W^{L-1}

$$\frac{\partial E}{\partial w^{(1)}_{1,1}} = -0.044\ (1) = -0.04\ ; \quad \frac{\partial E}{\partial w^{(1)}_{1,2}} = -0.044\ (2) = -0.09\ ;$$

$$\frac{\partial E}{\partial w^{(1)}_{2,1}} = -0.039\ (1) = -0.04\ ; \quad \frac{\partial E}{\partial w^{(1)}_{2,2}} = -0.039\ (2) = -0.08\ ;$$

$$\frac{\partial E}{\partial w^{(1)}_{3,1}} = -0.020\ (1) = -0.02\ ; \quad \frac{\partial E}{\partial w^{(1)}_{3,2}} = -0.020\ (2) = -0.04$$

We thus calculated the derivatives with respect to all weights using only values already calculated during forward navigation of the network, from the first layer to the last.

5.3 Final recipe

Thus, the errors $\delta^{(L)}_k$ and the weights $w^{(L)}_{k,m}$ of the next level are used to calculate the error of the previous level, so they are propagated backwards from the normal operation of an ff neural network. Notice how all the necessary values have already been calculated during the feed forward.

For the sake of completeness, we report the general formula which, after our journey, I hope will no longer be intimidating:

$$\delta^{(l-1)}_m \equiv \frac{\partial E}{\partial a^{(l-1)}_m} = \begin{cases} (y_m - t_m) & l = L \\ \dot{h}^{(l-1)}(a_m) \sum_k^{K^{(l)}} w^{(l)}_{k,m}\ \delta^{(l)}_k & l < L \end{cases} \qquad m \in \left[1, K^{(l-1)}\right] \tag{6}$$

Python code: We calculate the gradient using the self-differentiation of tensorflow.

```
# setup auto-differentiation
with tf.GradientTape() as tape:
    y_pred = model(x)                                    # Forward pass
    loss = tf.reduce_mean(tf.square(y_true - y_pred))    # MSE (L2)

# Compute gradient
grads = tape.gradient(loss, model.trainable_variables)

# show gradients for our datapoint (direct)
print("--- Input:", x.numpy(), "\n")
print("--- Gradients")
print(f"Layer 1:\n{grads[0].numpy().T}")
print("Output:", model.layers[0](x).numpy(), "\n")
print(f"Layer 2:\n{grads[1].numpy().T}")
print("Output:", y_pred.numpy(), "\n")
print("--- Label:", y_true.numpy(), "\n")
```

The output of the script confirms our calculations to an accuracy level of more than two decimal places.

```
--- Input: [[1. 2.]]

--- Gradients
Layer 1:
[[-0.04150072 -0.08300145]
 [-0.03751578 -0.07503156]
 [-0.01951589 -0.03903178]]
Output: [[0.46211717 0.8004991  0.935409  ]]

Layer 2:
[[-0.23709649 -0.41070867 -0.47992632]
 [-0.00169051 -0.00292837 -0.00342189]]
Output: [[0.48693424 1.1463418 ]]

--- Label: [[1.   1.15]]
```

6 Conclusions

In this article we have tried to introduce the well-known backpropagation algorithm by providing the theoretical elements necessary for understanding and a numerical example with which to compare the results obtained. The presentation is completed by a simple Python script with keras and tensorflow available on github.

References

Bishop, Christopher Michael and Hugh Bishop (2023). *Deep Learning - Foundations and Concepts*. Ed. by Springer Cham. 1st ed. ISBN: 978-3-031-45468-4. DOI: https://doi.org/10.1007/978-3-031-45468-4.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (Oct. 1986). "Learning Representations by Back-propagating Errors". In: *Nature* 323. Seminal paper that popularized backpropagation in neural networks., pp. 533–536.

Werbos, Paul J. (1974). "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences". First application of backpropagation to neural networks. PhD Dissertation. Harvard University.

Linnainmaa, Seppo (1970). "Taylor Expansion of the Accumulated Rounding Error". Introduces reverse mode automatic differentiation, foundational for backpropagation. Master's Thesis. University of Helsinki.

Minsky, Marvin and Seymour Papert (1969). *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press.

Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Cornell Aeronautical Laboratory. Report no. VG-1196-G-8. Spartan Books. URL: https://books.google.it/books?id=7FhRAAAAMAAJ.

Kelley, Henry J. (1960). "Gradient Theory of Optimal Flight Paths". In: *Journal of the Institute of the Aeronautical Sciences* 30.10. Early gradient method in control, precursor to backprop., pp. 947–954.

Mcculloch, Warren and Walter Pitts (1943). "A Logical Calculus of Ideas Immanent in Nervous Activity". In: *Bulletin of Mathematical Biophysics* 5, pp. 127–147.