A Primer on Backpropagation with a Numerical Example, Diagrams and Python Code

Nicola Santi

July 26, 2025

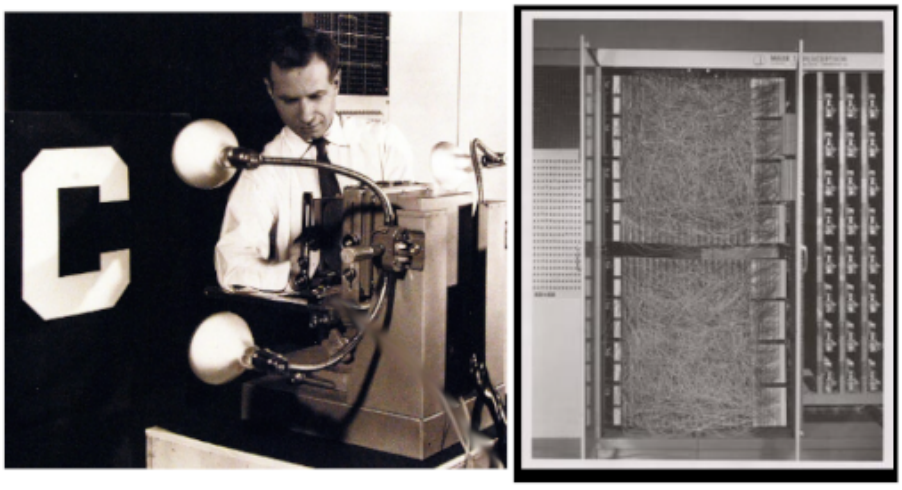
Abstract

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 learned. A Python script with Keras and TensorFlow to verify the calculations performed completes the exposition (available on GitHub).

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, no one 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.

Although the first simple artificial neural network dates back to Mcculloch and Pitts (1943), it was with Rosenblatt (1962) that the theory and hardware needed to implement the first neural network called *perceptron* were developed. The hardware needed to implement it is depicted in the photos below where we can also see Charles Wightman (project engineer) adjusting Mark I Perceptron:



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 by 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 of *perceptron* would also apply to neural networks with more than one layer (deep learning).

This widespread belief led to a suspension of interest, funding and research in 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 differentiable, 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), 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 forward pass (from the first layer to the last) to calculate the gradient, which saves resources and makes it possible to handle 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 through some elementary calculations on the example model, followed by a short Python script to verify their correctness.

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:

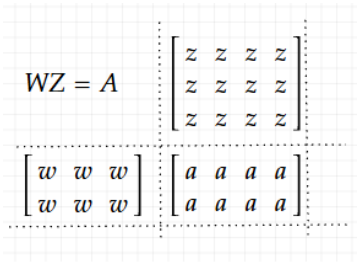


Figure 1: Matrix multiplication

The advantage is the immediate visualization of the dimensions of the resulting matrix, which will have as many rows as W and as many columns as Z. Recall 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 $W\boldsymbol{z} = \boldsymbol{a}$:

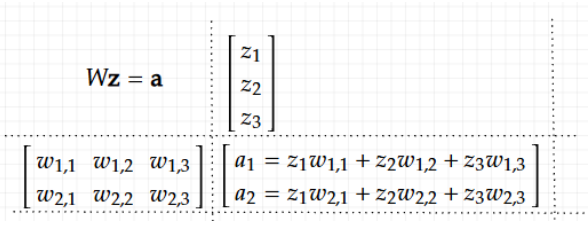


Figure 2: Matrix and vector multiplication

[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 $\boldsymbol{z}$ contributes to determining each element of $\boldsymbol{a}$. This awareness will be useful in one of the final sections of this paper (5.1). Moreover, this is precisely why layers involving matrix multiplication are called dense: each input element influences every output element.

Finally, each a_k being the sum of products between elements of $\boldsymbol{z}$ 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 it first applies a linear transformation W and then a nonlinear transformation $h()$. The result is used by the next layer as input (hence the feedforward meaning), see Figure below.

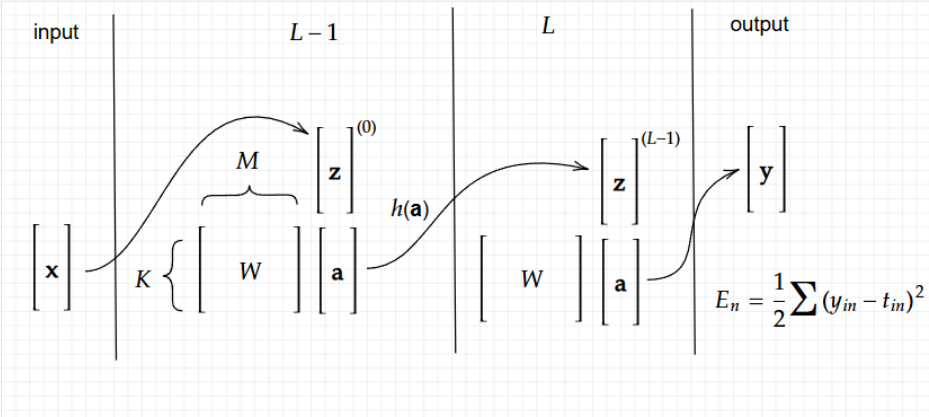


Figure 3: Simple feedforward network

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 $\mathbf{a} \in \mathbb{R}^K$.

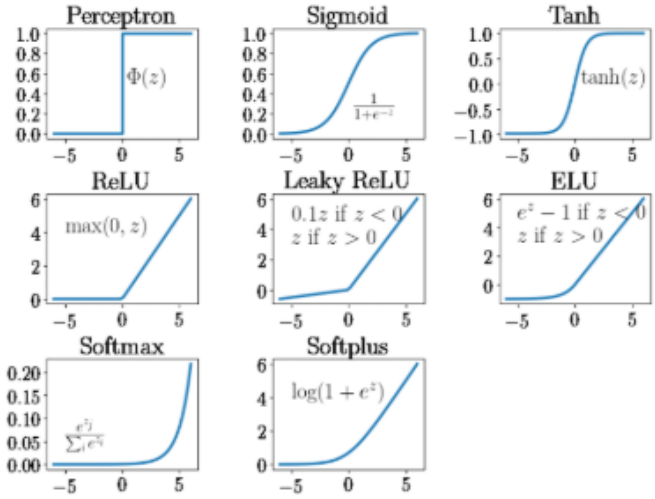


Figure 4: Some common activation functions

A potentially nonlinear transformation called activation function is then applied to vector $\mathbf{a}$. Common examples of this kind of function are softmax, 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 (also known as error 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 layer must match the next layer input. The last three equalities will be particularly useful in paragraph 4.2;

Note

For didactic purposes, the last layer L has no activation function or, equivalently, uses the identity function $h(\boldsymbol{a}) = \boldsymbol{a}$. We will return to this point in section **??**. For similar reasons, each layer has no bias.

The superscripts in parentheses, as mentioned, indicate the layer to which each variable refers: they are somewhat inconvenient, so, for simplicity, in case of omission it will be understood that we are referring to the current layer 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 definition

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

The code is very simple: the exclusion of the bias and the activation function of the last layer, which coincides with the identity function (as if it were not there), are worth noting.

Please remember that the source code is available on GitHub.

4 Four simple derivatives (in fact, only two)

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

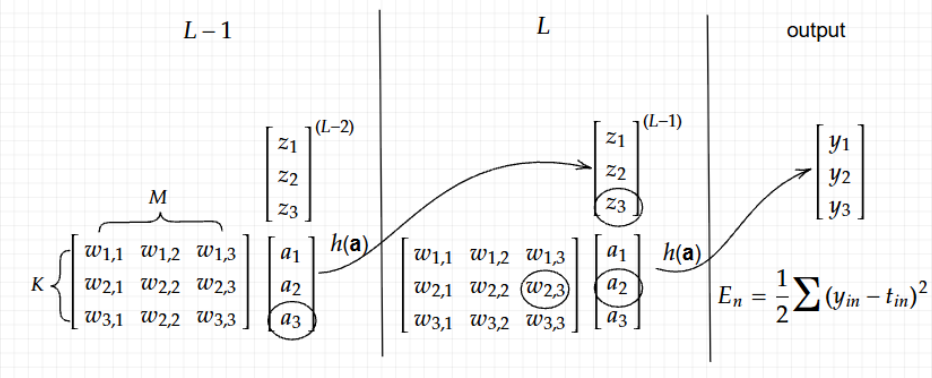


Figure 5: Derivatives of $\boldsymbol{a}$ with respect to the current layer L and the previous one

4.1 Inside 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 the derivative of a_2 with respect to $w_{2,3}$ is simply z_3. Or, conversely, the derivative of a_2 with respect to z_3 is $w_{2,3}$.

Note: equation (2), upon 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 $\boldsymbol{a}^{l-1}$ that we will calculate in the next paragraph.

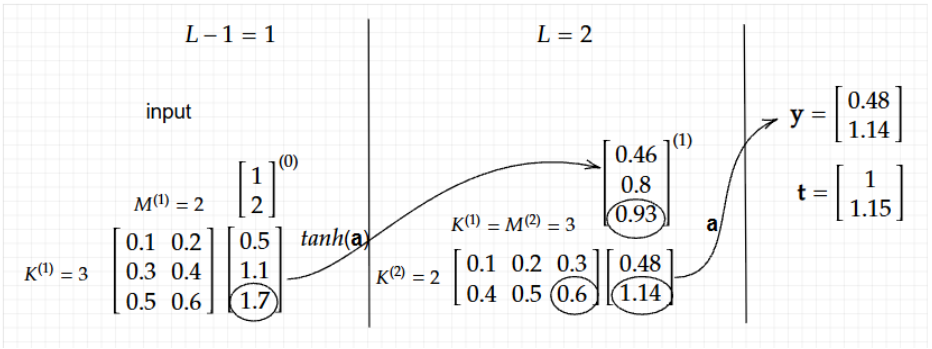


Figure 6: A numerical example

Our example: to refine our 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 $\boldsymbol{t}$ is the truth value for a certain data point. All values are approximated to two decimal places. Now, referring to layer 2, we calculate the partial derivatives of a_2.

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

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

Python code: we can recreate our example by initializing the weights of each layer, instantiating the x input tensor and the t label.

Parameters initialization

```
# 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, derivative (2) already links the current layer to the previous one, because $\mathbf{z}^{(l-1)}$ is both the previous layer output and the current layer input. But we need to be linked to previous $\boldsymbol{a}^{l-1}$. 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 $\dot{h}^{l-1}()$, the derivative of activation function $h^{l-1}()$. If you are surprised by the use of the same index m to refer to vectors of different layers, $z^{(l-1)}$ after all lies in layer l while $a^{(l-1)}$ lies in $l-1$, just take a look at Figure 3 and remember that $M^{(l)} = K^{(l-1)}$.

Our example: Again, using Figure 6, we calculate the derivative of the third component of the first-layer output $z_3^{(1)}$ (which is also the third component of the second-layer input) with respect to $a_3^{(1)}$. Remember that, in our example, the activation of layer $L-1$ is tanh() and that the derivative of tanh() is precisely $1 - \tanh^2()$:

$$\frac{\partial z_3^{(1)}}{\partial a_3^{(1)}} = \dot{\tanh}(1.7) = 1 - \tanh^2(1.7) = 1 - (0.93)^2 = 0.14$$

We have developed the necessary formulas to finally calculate the derivative of **each single component** of $\boldsymbol{a}$ with respect to each $a_m^{(l-1)}$ of the previous layer using equations (2) and (3):

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

As in (3), m and k refer both to layer l.

Our example: we simply apply equation (4)

$$\frac{\partial a_2^{(2)}}{\partial a_3^{(1)}} = (0.14)\ 0.6 = 0.084$$

In conclusion, we will only need a couple of derivatives from this paragraph, namely equations (1) and (4).

5 Backpropagation algorithm

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 $w_{k,m}$.

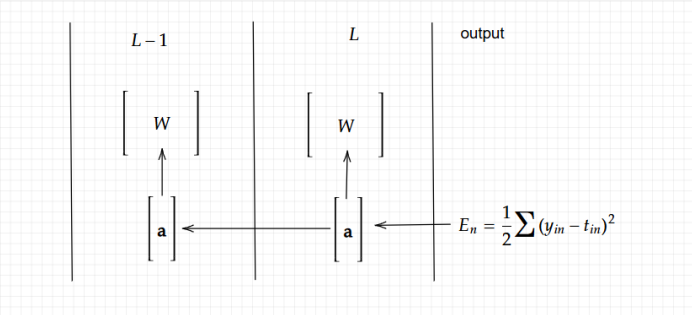


Figure 7: Backpropagation moves from layer to previous one using $\boldsymbol{a}$ derivatives and then computes gradient with respect to W.

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

5.1 Meet δ, the propagated error

The full name of the algorithm that is the object of this article is *backward propagation of errors*, referring precisely to the errors of the model output, which are also the individual components of the loss function. Let us meet $\boldsymbol{\delta}$ for the last layer:

$$\delta_k^{(L)} = y_k - t_k \tag{5}$$

You could note that it is precisely the difference between the k-th component of the model output $\boldsymbol{y}$ and the correct value $\boldsymbol{t}$, so we could call $\boldsymbol{\delta}$ simply *errors* and define it formally for each layer:

$$\delta_k^{(l)} \equiv \frac{\partial E}{\partial a_k^{(l)}} \tag{6}$$

It is simply the partial derivative of the loss function with respect to a component of $\boldsymbol{a}$. Remember from Figure 7 that backpropagation uses these intermediate derivatives to reach its ultimate goal: the derivative of E with respect to each W.

Finally, that equation (6), in the last layer, coincides with equation (5) is easily demonstrated:

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

Please note:

As mentioned, in our simplified model the last layer has no activation function: if there were one, it would have to be taken into account in exactly the same way as equation (3), but the interpretation of $\boldsymbol{\delta}$ as closely related to the error measure would remain valid.

Our example: Let's calculate the error $\boldsymbol{\delta}^L$ of our example, please return to Figure 6:

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

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

Equation (4) connects each component of $\boldsymbol{a}^{(l)}$ with each component of $\boldsymbol{a}^{(l-1)}$, but the calculation of $\boldsymbol{\delta}^{(l)}$ for the layers preceding the last one requires us to give additional thought: E depends on all the components of $\boldsymbol{a}^{(l)}$ (not just one), so we will have to take into account (by adding them up) all these variations.

Remembering paragraph 2 and with the help of Figure 6, we obtain:

$$\delta_m^{(L-1)} = \frac{\partial E}{\partial a_m^{(L-1)}} \;\; = \sum_k^{K^{(L)}} \underbrace{\frac{\partial z_m^{(L-1)}}{\partial a_m^{(L-1)}} \frac{\partial a_k^{(L)}}{\partial z_m^{(L-1)}}}_{(4)} \underbrace{\frac{\partial E}{\partial a_k^{(L)}}}_{(6)}$$

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

We have used equations (4) and (6) where indicated. As before, m and k refer to layer L.

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 $\boldsymbol{\delta^{L-1}}$

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

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

$$\delta_m^{(l-1)} \equiv \frac{\partial E}{\partial a_m^{(l-1)}} \;\; = \begin{cases} (y_m - t_m) & l = L \\ \dot{h}^{(l-1)}(a_m) \sum_k^{K^{(l)}} \; w_{k,m}^{(l)} \; \delta_k^{(l)} & l < L \end{cases} \qquad m \in \left[1, M^{(l-1)}\right] \tag{7}$$

5.2 Conclusion of propagation

Having the (partial) derivatives with respect to $\boldsymbol{a}^{(l)}$ for each layer, it is rather simple to find the derivatives of E with respect to each parameter $w^l_{k,m}$ and thus ultimately achieve the goal of calculating the gradient of the loss function with respect to the parameters.

We complete the calculation for the last layer by deriving $\boldsymbol{a}$ with respect to each element W. We are helped by equations (1) and (6).

$$\frac{\partial E}{\partial w^L_{k,m}} = \frac{\partial E}{\partial a^L_k}\frac{\partial a^L_k}{\partial w_{k,m}} \;=\; \delta^L_k \; z^{(l-1)}_m$$

Our example: compute the derivative for each $w_{k,m}$ of the last layer $L = 2$

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

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

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

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

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

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

Please note that we have calculated the derivatives with respect to all weights using only values already computed during the forward pass of the network, from the first layer to the last.

6 Checking the calculations

We add code to instruct our script to perform the gradient calculation in order to check the accuracy of our results. When comparing, we take into account the rounding to two decimal digits used in our examples.

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

Gradient calculation

```python
# 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, see the next figure, 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]]
```

7 Conclusions

In this article, we illustrated the backpropagation algorithm step by step by placing theory side by side with a small neural network used to refine intuition and carry out calculations. We did not skimp on graphs and insights, going as far as implementing a simple Python script using 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.