Training Neural Networks

Alberto Zaupa

27 April 2022

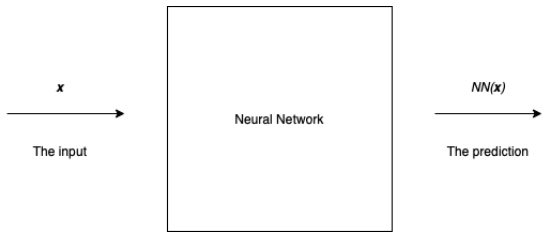
1 Table of contents

2 The Neural Network model

An artificial *Neural Network* is a computational system, inspired by the biology of the animal brain, composed of individual units of computation, called *neurons*, that are organized into *layers*, which are then connected together in order to form the whole architecture. Neural Networks are a very powerful and general machine learning model.

To begin with, we are going to describe the *Neural Network* model as a composition of systems, of which we are going to define the input/output relationships.

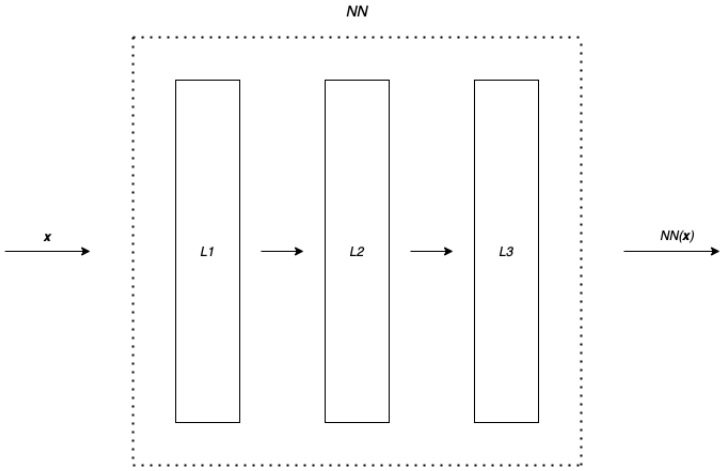
2.1 A top-down characterization



A *NN* is a system that maps vectors, which hold scalar values that correspond to *features* (relevant quantities) extracted from the phenomenon we interested in, to other vectors, which hold *synthetic information* (for example a

future prediction, or a classification) extracted on the basis of the input data, through a non-linear transformation:

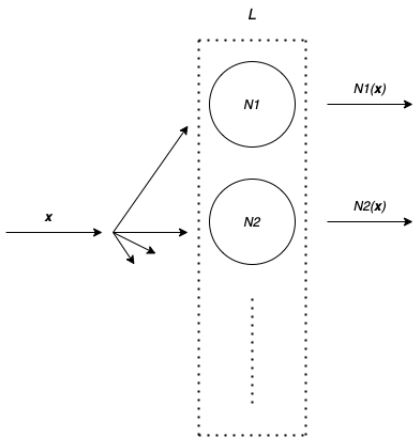
$$NN(\mathbf{x}) = \psi(\mathbf{x}).$$



If we open the box of the neural network, we find that inside it is a *series connection* of other systems, which are the *layers* of the neural network, each of which performs a computation based on the input it receives, and forwards its output to the subsequent layer.

Each layer is a system characterized by a mapping from an input vector, to an output vector (not necessarily of the same dimension), through, again, a non-linear transformation:

$$L(\mathbf{x}) = \phi(\mathbf{x})$$

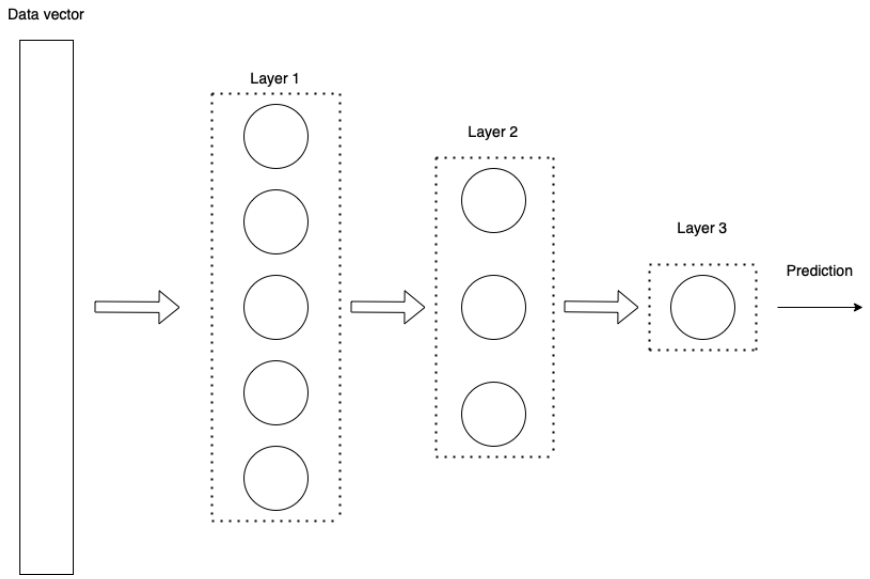


If we then inspect the layers of the neural network, we find that they consist of a *parallel connection* of simpler systems called *neurons*, each of which takes in the same input vector to the layer and produces a scalar value.

The neurons are systems that map vector from nD space to real values through, again a non-linear function:

$$N(\mathbf{x}) = f(\mathbf{x})$$

That is the complete picture of how a neural network works when we think of it in terms of systems:



We are now going to describe in details the computation performed by a neural network with a bottom-up approach, by starting from the smallest elements of the network, the neurons.

2.2 Neurons

Artificial neurons are the basic units of computation of a *NN*. They are an approximate mathematical model of the biological neuron.

Biological neurons take in electric signals from other neurons, and produce as an output a signal through a computation that we model as a nonlinear transformation of a weighted sum of the input signals.

The linear combination of the input signals can be represented as a scalar product of a vector of weights and the vector of the input values. The scalar product added together with an offset is then given as input to a non-linear *activation* function.

$$N(\mathbf{x}) = f(\mathbf{w}^T\mathbf{x} + b)$$

One of the most used activation functions is the *sigmoid* function:

$$\sigma(u) = \frac{1}{1 + e^{-u}}$$

Another common activation function, that has now become the go-to non-linearity, is ReLU, *Rectified Linear Unit*:

$$ReLU(u) = max(0, u)$$

2.3 Layers

The architectural aspect of the *NN* model that makes it so powerful is its organization into layers of neurons. From the inside, a layer is a collection of neurons that each receive the same input vector and produce a scalar value, therefore the length of the output vector of a layer is equal to its number of neurons. The activation function of the neurons of the a layer is generally the same.

We can model the relationship between the input vector and the output vector of a layer the following way:

$$Layer(x) = f(W^T x + b)$$

where W is the matrix that combines the weight vectors of the neurons in the layer, x is the input vector, w_0 is the vector of biases, and $f(\cdot)$ is the activation function of the neurons of the layer, applied element wise to the matrix transformation.

A *deep NN* is built as a sequence of layers, each of which can have an arbitrary number of neurons, that are connected sequentially. Therefore the first layer takes in a input vector of features and produces an output vector that is then given as input to the following layer, which then passes on its output to the following layer, and so on.

Of course the last layer of the network must produce a vector that has the same dimension of our desired output. If we are doing one-dimensional regression, the desired output is a vector of dimension 1, therefore the last layer must have a single neuron.

2.4 Other architectures

The organization of layers just described is relevant to the more basic model of *NN*, that is called *fully-connected feed-forward Neural Network*. In this architecture data flows sequentially, and every neuron of a layer is connected to every neuron of the following layer.

This architecture is not completely general because it does not include the notion of *state*. The output of such a network can only depend of the current input, but in some applications we want the output to also depend on the internal state of the system.

Recurrent Neural Networks introduce the notion of state by allowing the outputs of some layers to be connected to the inputs of some previous layers.

Fundamentally, the difference between recurrent *NNs* and feed-forward *NNs* is the same as the difference between combinatorial circuits and sequential circuits in logic design.

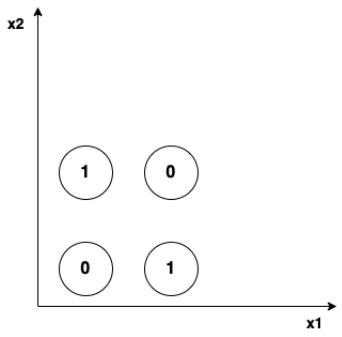
3 Why Neural Networks

Although the basic components of Neural networks are relatively simple, as a whole they are complex computational systems, and because of that it is not easy to fully understand their most inner workings. Also, as we shall see, training a neural network is not a completely straightforward process and it requires quite a bit of heuristics. For these reasons although the mathematical models of neural networks and the techniques used to train them have been known for at least 40 years, for a long time they were not widely used.

In recent years, though, they have come at the forefront of machine learning applications because, thanks to the increase in computational power and to the refinement of some learning techniques, they have been able to surpass every other machine learning model in many problem domains.

As already stated, the characteristic of the *NN* model that makes it so powerful is its organization into layers.

3.1 Solving XOR with a NN



Let's suppose that we wanted to train a classifier to learn the input/output mapping shown in the picture above. This problem is commonly known in the literature as the *XOR problem*, because it requires us to separate some points that are labeled in a way that reminds of the truth table of the XOR function.

Our data points are:

- $\mathbf{x} = [0, 0]$ $\mathbf{y} = 0$
- $\mathbf{x} = [0, 1]$ $\mathbf{y} = 1$
- $\mathbf{x} = [1, 0]$ $\mathbf{y} = 1$
- $\mathbf{x} = [1, 1]$ $\mathbf{y} = 0$

If we were to use a linear model, it would not be possible to learn such a mapping. Intuitively, the reason why that is the case is that there is no linear transformation that can map our data points into a feature space where the 1-labeled vectors can be separated from the 0-labeled vectors through an hyperplane.

We'll solve this problem with a 2-layered neural network and we'll show how the non-linear activations play a crucial role.

We'll straightaway provide the parameters of the network that correspond to a possible solution. In a real world scenario, these parameters would be the outcome of the training process.

Let's suppose the first layer of our network is defined by the weight matrix W_1 and the bias vector b_1:

$$W_1 = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

$$b_1 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

$$X = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

Let's compute $W_1^T X$:

$$W_1^T X = \begin{pmatrix} 0 & 1 & 1 & 2 \\ 0 & 1 & 1 & 2 \end{pmatrix}$$

Let's now add the bias vector:

$$W_1^T X + b_1 = \begin{pmatrix} 0 & 1 & 1 & 2 \\ -1 & 0 & 0 & 1 \end{pmatrix}$$

This linear transformation was not able to move our data points in a way that allows us to separate them with an hyperplane, as they all lay on the same line $x_2 - x_1 + 1 = 0$.

Let's now feed our transformed data points to a ReLU activation layer:

$$ReLU(W_1^T X + b_1) = \begin{pmatrix} 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

We can now clearly see that it is possible to separate with an hyperplane the 1-labeled data points from the 0-labeled data points. A candidate hyperplane is $x_2 - \frac{5}{8}x_1 + \frac{1}{2} = 0$.

The output layer of the network has a single neuron that maps from 2D space to 1D space, and its activation function is $f(u) = max(0, u)$. If we chose the hyperplane previously stated, these would be the parameters of the output layer:

$$W_2 = \begin{pmatrix} -\frac{5}{8} \\ 1 \end{pmatrix}$$

$$b_2 = \frac{1}{2}$$

3.2 Takeaway from XOR

In the example we have just seen, the fundamental element of our model that allowed it to solve the problem was the non-linearity of the activation functions. In general a neural network with at least two layers, with a sufficient amount of neurons, and with non-linear activation functions, is able to approximate any function $f^*(\cdot)$ to an arbitrary degree of precision.

Interestingly in building the approximation to the desired function we did not assume anything about the shape of f^*: this is because our network was able to learn the necessary parameters to allow the first layer to map the original feature space into a new feature space where the approximation could be performed through a simple linear function. We can imagine that stacking up more layers of neurons allows the network to learn extremely complicated mappings from the initial feature space to more information-dense feature spaces.

This fact is extremely important: neural networks can autonomously extract from the training data the necessary features to allow for a simple (linear) solution to the problem at hand.

This is the fundamental property of neural networks that makes them so powerful and appealing because it allows us to spend less time on the very difficult process of feature engineering.

4 Training a Neural Network

We have seen that a neural network is a computational system that can extract some desired *synthetic information* from input data, through a learned mapping $f^*(\cdot)$. Initially, though, $f^*(\cdot)$ is unknown, therefore we need a procedure for allowing the *NN* to learn it.

For example if we are using a neural network to determine whether some images contain cats or not, in the beginning the neural network will be completely incapable of recognizing cats, because the *rule* that allows to determine whether an arbitrary image contains a cat or not, is unknown. In order for the network to be able to recognize cats correctly, we need to *train* it.

4.1 The training setup

To train our neural networks we'll take the same approach that is commonly used in the context of supervised learning: we'll define a *loss function* that, given an input data point and the corresponding expected output of the *NN*, determines how wrong the prediction of the network is.

The choice of the loss function is critical and must take into consideration the peculiarities of the problem domain. For example, a loss function that can

be used in the context of scalar regression should not be used in the context of classification.

We'll consider a simple loss function that is suited to the context of scalar regression, which undergoes the name of *squared error loss*:

$$Loss(\mathbf{x}, \mathbf{y}) = (NN(\mathbf{x}) - \mathbf{y})^2$$

where $\mathbf{x}$ is an input feature vector, and $\mathbf{y}$ is the expected output of the network. The loss is the squared distance between the prediction and the corresponding expectation.

4.2 Optimizing for the loss function

Now that we have defined a way to determine how unhappy we are about the predictions of our neural network, we would like to find the values of the parameters of the model, the weights of the network, that minimize the value of the loss function, calculated on the training data.

We cannot derive a closed-form mathematical expression for the optimal values of the weights, as the prediction of our neural network is a composition of non-linear functions, so in order to do that we would need to solve a non-linear system. We have to resort to numerical optimization method and, as it is common in the field of machine learning, we'll chose the simplest of them all: gradient descent.

4.3 Gradient descent

Gradient descent is a simple optimization technique that given a function

$$f(\mathbf{x}) : D \subset R^n \rightarrow I \subset R$$

has the goal of finding the value for $\mathbf{x}$ that minimizes f .

The technique is based on a fundamental result of multi-dimensional calculus that states that given a *f*, for example the one above, the following holds:

$$f(x_0 + \delta) = f(x_0) + \delta \cdot \nabla_f(x_0), \qquad ||\delta|| \rightarrow 0$$

As a consequence of this result, the gradient of a function at a particular point corresponds to the direction of maximum increase of the function while remaining close to the same point, and the negative gradient corresponds to the direction of maximum decrease.

Gradient descent provides an iterative algorithm that allows us to start from a certain value of the weights of our model, compute the gradient of the loss function given those values of the weights, update the weights by taking a small step in the direction opposite to that of the gradient, and repeat the process until we converge to the value of the weights that minimizes the loss function.

$$W(t+1) \leftarrow W(t) - \eta \nabla_{f,W}(W(t))$$

If the loss function is convex and our model is linear, given enough iterations and a small enough η, we are guaranteed to get the value of the weights that corresponds to the minimum value of the loss function. But in the case of neural networks, our model is not linear, so the only guarantee we have is that we will reach a local minimum, instead of the global minimum.

It is important to state that although we are interested in moving towards the global minimum, we should stop before reaching it. The loss function is a measure of the performance of the model on the *training data*, therefore a minimum value of the loss function probably leads to the *overfitting* of the model, which will then not be able to generalize well on unseen data.

5 Backpropagation

In order to train our *NNs* we are going to use gradient descent to iteratively update the values of the weights of each layer, until we obtain a sufficiently low value of the loss function.

Computing $\nabla_{Loss,W}$ is not easy when the neural networks have many layers, of many neurons each: imagine having to compute the partial derivative of the loss function with respect to the 1000th weight in the first of 10 layers in a neural network.

In order to avoid having to keep track of many indices, we are going to use vector calculus to keep the mathematical expressions more compact. We are also going to make extensive use of the chain rule, which states that given $y = f(a)$, where $a = g(b)$:

$$\frac{\partial y}{\partial b} = \frac{\partial y}{\partial a}\frac{\partial a}{\partial b} = f'(g(b))g'(b)$$

After deriving the mathematical expression for the partial derivative of the loss function with respect to the weights of an arbitrary layer, we'll discuss *Backpropagation*, which is an algorithm that allows us to compute our desired derivatives while keeping the computational complexity under control.

This is not an easy section to read through, mainly because the notation will quickly become quite complicated. So do not worry if you need to read the section many times before you actually understand it.

5.1 Notation

Given a neural network with L layers of neurons, for the layer of index l we define the following parameters:

- m_l the length of the input vector to the layer
- n_l the length of the output vector of the layer
- a_{l-1} the input vector to the layer, which, except for the first layer, is equal to the output vector of the previous layer. It has dimensions m_l *x 1*

- W_l the weight matrix of the layer, which has dimensions m_l *x* n_l
- b_l the vector of biases, of dimensions n_l *x 1*
- $f_l(\cdot)$ the activation function of the layer
- z_l the *pre-activation* vector, which has dimension n_l *x 1.*

$$z_l = W_l^T a_{l-1} + b_l$$

- a_l the *activation* vector. It is the output of the layer and it has dimensions n_l *x 1.*

$$a_l = f_l(z_l)$$

5.2 An introductory example

We'll begin by considering a very simple neural network that takes in scalar values and that has two layers with just one neuron each. It follows that all the weight matrices and bias vectors are just scalar values. Our loss function will be *MSE*:

$$Loss(x, y) = (NN(x) - y)^2$$

We'll try to determine the mathematical expressions of the partial derivatives of the loss function with respect to the two weights of the network, ignoring the biases and their contribution in order to concentrate on a lesser number of details.

First of all, let's compute $\frac{\partial Loss}{\partial W_2}$. It is clear that we need to use the chain rule because the loss function is not directly a function of W_2. $Loss(\cdot)$ in fact is a function of a_2, which then is a function of z_2, which finally is a function of W_2:

$$Loss(a_2) = (a_2 - y)^2$$
$$a_2 = f_2(z_2)$$
$$z_2 = W_2^T a_1$$

because W_2 and a_1 are scalars, $W_2^T a_1$ is just a simple product. So now by applying the chain rule we get:

$$\frac{\partial Loss}{\partial W_2} = \frac{\partial Loss}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial W_2} \Leftrightarrow$$

$$\frac{\partial Loss}{\partial W_2} = 2(a_2 - y)f_2'(z_2)a_1$$

Now that we've got the partial derivative of the loss function with respect to the weight in the second layer, let's derive the partial derivative of the loss

function with respect to the weight in the first layer. Again we'll have to use the chain rule because the loss function is not directly a function of W_1.

$$Loss(a_2) = (a_2 - y)^2$$
$$a_2 = f_2(z_2)$$
$$z_2 = W_2 a_1$$
$$a_1 = f_1(z_1)$$
$$z_1 = W_1 x$$

therefore:

$$\frac{\partial Loss}{\partial W_1} = \frac{\partial Loss}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial W_1} \Leftrightarrow$$

$$\frac{\partial Loss}{\partial W_1} = 2(a_2 - y)f_2'(z_2)W_2 f_1'(z_1)x$$

We have computed the whole gradient of the loss function with respect to the weights of the network. Now we could update the weights by taking a small step along the gradient, and by iterating this process the neural network would learn the unknown mapping from the feature space of our input data to the desired output domain!

Before moving on to generalize the approach taken in computing the gradient by using the chain rule, let's underline some useful details of the formulas we have derived.

$$\frac{\partial Loss}{\partial W_2} = \frac{\partial Loss}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial W_2}$$

$$\frac{\partial Loss}{\partial W_1} = \frac{\partial Loss}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial W_1}$$

We can observe that the term $\frac{\partial Loss}{\partial a_2}\frac{\partial a_2}{\partial z_2}$ is repeated in the right sides of the two equations. Actually the expression for $\frac{\partial Loss}{\partial W_2}$ is pretty much repeated in the expression for $\frac{\partial Loss}{\partial W_1}$, the only difference is that the term $\frac{\partial z_2}{\partial W_2}$ is changed for $\frac{\partial z_2}{\partial a_1}$.

This fact suggest that an algorithm for computing $\nabla_{Loss,W}$ that was concerned with managing computational complexity, could achieve that goal by propagating some terms *backwards through the network*, expanding them at each layer.

5.3 Computing $\nabla_{Loss,W}$, the general case

Now that we have got a feel for how to go about computing partial derivatives for the weights of our neural network, it is time to generalize the approach.

We are going to consider a neural network with L layers, and we are going to refer to the notation defined previously. Be careful that for the layer of index l, $m_l = n_{l-1}$ and $n_l = m_{l+1}$.

Let's compute the partial derivative of the loss function with respect to the weights in the layer l: $\frac{\partial Loss}{\partial W_l}$. We are again not going to consider the bias vectors, but the same reasoning could be applied to compute $\frac{\partial Loss}{\partial b_l}$ (the expression for this partial derivative is actually simpler).

The loss function is not directly a function of W_l so we need to use the chain rule. We are going to first determine the sequence of terms in the computation that depend on each other, starting from the loss function all the way down to the weights in the layer l.

$Loss(\cdot)$ is directly a function of a_L, a_L is directly a function of z_L, which is then a function of a_{L-1}, which is a function of z_{L-1}, which is a function of a_{L-2}, and so on until we get to a_l, which is a function of z_l, which finally is a function of W_l. Now through the chain rule we can determine an expression for $\frac{\partial Loss}{\partial W_l}$:

$$\frac{\partial Loss}{\partial W_l} = \frac{\partial Loss}{\partial a_L}\frac{\partial a_L}{\partial z_L}\frac{\partial z_L}{\partial a_{L-1}}\frac{\partial a_{L-1}}{\partial z_{L-1}}\frac{\partial z_{L-1}}{\partial a_{L-2}}...\frac{\partial a_l}{\partial z_l}\frac{\partial z_l}{\partial W_l}$$

Let's now determine the values of these partial derivatives. We know that $\frac{\partial Loss}{\partial a_L}$ is just $Loss'(a_L)$. The terms $\frac{\partial a}{\partial z}$ are clearly equal to the derivative of the activation function of the corresponding layer: $\frac{\partial a_L}{\partial z_L} = f'_L(z_L)$, $\frac{\partial a_{L-1}}{\partial z_{L-1}} = f'_{L-1}(z_{L-1})$, and so on. The term $\frac{\partial z_l}{\partial W_l}$ is simply a_{L-1}, because the pre-activations are a linear function of the weights. And finally the terms $\frac{\partial z}{\partial a}$, which are the derivatives of the pre-activations of a layer with respect to the input of that layer, are just the weights of the corresponding layer, as again the pre-activations are a linear function of the input of the layer. Therefore $\frac{\partial z_L}{\partial a_{L-1}} = W_L$, $\frac{\partial z_{L-1}}{\partial a_{L-2}} = W_{L-1}$, and so on.

$$\frac{\partial Loss}{\partial W_l} = Loss'(a_L)f'_L(z_L)W_L f'_{L-1}(z_{L-1})W_{L-1}...f'_l(z_l)a_{l-1}$$

or equivalently:

$$\frac{\partial Loss}{\partial W_l} = \frac{\partial Loss}{\partial a_L}\frac{\partial a_L}{\partial z_L}W_L\frac{\partial a_{L-1}}{\partial z_{L-1}}W_{L-1}...\frac{\partial a_l}{\partial z_l}a_{l-1}$$

So we have determined all the terms that contribute to the partial derivative of the loss function with respect to the weights of a generic layer, and we were able to determine what each of those terms corresponds to. But we have to be careful that the expression:

$$\frac{\partial Loss}{\partial W_l} = \frac{\partial Loss}{\partial a_L}\frac{\partial a_L}{\partial z_L}W_L\frac{\partial a_{L-1}}{\partial z_{L-1}}W_{L-1}...\frac{\partial a_l}{\partial z_l}a_{l-1}$$

is not correct because its terms are vectors and matrices, and we have still not concerned ourselves with making sure that all the dimensions match correctly. We'll do it now. First of all we'll determine the shape of each of those terms:

- $Loss'(a_L)$. The loss function produces a scalar value, but its input is a vector of dimensions n_L *x 1*. So when taking its derivative, we are going to end up with a n_L *x 1* vector that holds the values of the derivative of the loss with respect to each of the terms in a_L.

$$Loss'(a_L) = \begin{pmatrix} \frac{\partial Loss}{\partial a_{L,1}} \\ \frac{\partial Loss}{\partial a_{L,2}} \\ ... \\ \frac{\partial Loss}{\partial a_{L,n_L}} \end{pmatrix}$$

- $f'_l(z_l)$. These terms require us to compute the derivative of the output vector of a layer, of dimensions n_l *x 1*, with respect to the pre-activations vector of the same layer, again of dimensions n_l *x 1*. This derivative is a n_l *x* n_l diagonal matrix, because for each term of a_l, we have to compute its derivative with respect to all of the terms of z_l. But $a_{l,i}$ is just a function of $z_{l,i}$ and not of any other of the terms of z_l. Therefore all the terms of the matrix of derivatives that are not on the diagonal will be 0.

$$f'_l(z_l) = \begin{pmatrix} \frac{\partial a_{l,1}}{\partial z_{l,1}} & 0 & 0 & ... \\ 0 & \frac{\partial a_{l,2}}{\partial z_{l,2}} & 0 & ... \\ ... & & & \\ ... & 0 & 0 & \frac{\partial a_{l,n_l}}{\partial z_{l,n_l}} \end{pmatrix}$$

- W_l. As we know the weight matrix of layer l has dimensions m_l *x* n_l, but because $m_l = n_{l-1}$, we'll say that W_l has dimension n_{l-1} *x* n_l

- a_{l-1}. Finally we get to the last term, which as already stated has dimensions n_{l-1} *x 1*.

We also know that $\frac{\partial Loss}{\partial W_l}$ is a n_{l-1} *x* n_l matrix. Let's see how the dimensions should match.

We start from $\frac{\partial Loss}{\partial a_L}$, which is n_L *x 1*, and we need to multiply it by the n_L x n_L matrix that is $\frac{\partial a_L}{\partial z_L}$, in order to get $\frac{\partial Loss}{\partial z_L}$ which is n_L *x 1*, therefore we need to perform the multiplication the following way:

$$\frac{\partial Loss}{\partial z_L} = \frac{\partial a_L}{\partial z_L}\frac{\partial Loss}{\partial a_L}$$

Then we need to compute $\frac{\partial Loss}{\partial a_{L-1}}$, which is n_{L-1} *x 1*. Because $\frac{\partial z_L}{\partial a_{L-1}}$ is the n_{L-1} *x* n_L weight matrix, we get:

$$\frac{\partial Loss}{\partial a_{L-1}} = W_L(\frac{\partial a_L}{\partial z_L}\frac{\partial Loss}{\partial a_L})$$

Then we need $\frac{\partial Loss}{\partial z_{L-1}}$, and by following the same reasoning for $\frac{\partial Loss}{\partial z_L}$, we get:

$$\frac{\partial Loss}{\partial z_{L-1}} = \frac{\partial a_{L-1}}{\partial z_{L-1}}(W_L\frac{\partial a_L}{\partial z_L}\frac{\partial Loss}{\partial a_L})$$

which is a n_{L-1} *x 1* vector. By proceeding in this way we get to the layer l, knowing the value of $\frac{\partial Loss}{\partial a_l}$, which is n_l *x 1*. First let's compute $\frac{\partial Loss}{\partial z_l}$:

$$\frac{\partial Loss}{\partial z_l} = \frac{\partial a_l}{\partial z_l}\frac{\partial Loss}{\partial a_l}$$

.

Now that we have $\frac{\partial Loss}{\partial z_l}$ we can compute $\frac{\partial Loss}{\partial W_l}$. Because the final term should be n_{l-1} *x* n_l, and $\frac{\partial Loss}{\partial z_l}$ is n_l *x 1*, we get:

$$\frac{\partial Loss}{\partial W_l} = a_{L-1}(\frac{\partial Loss}{\partial z_l})^T$$

We finally know every detail about how to compute $\frac{\partial Loss}{\partial W_l}$. Let's write the whole expression in a single line:

$$\frac{\partial Loss}{\partial W_l} = a_{L-1}(\frac{\partial a_l}{\partial z_l}...W_{L-1}\frac{\partial a_{L-1}}{z_{L-1}}W_L\frac{\partial a_L}{\partial z_L}\frac{\partial Loss}{\partial a_L})^T$$

comparing it with the expression we had initially computed without bothering to match the dimensions, we see that the correct one is pretty much equal to the incorrect one, just written backwards:

$$\frac{\partial Loss}{\partial W_l} = \frac{\partial Loss}{\partial a_L}\frac{\partial a_L}{\partial z_L}W_L\frac{\partial a_{L-1}}{\partial z_{L-1}}W_{L-1}...\frac{\partial a_l}{\partial z_l}a_{l-1}$$

Now that we know how to compute the partial derivative of the loss function with respect to the weights of a particular layer l, we can compute the partial derivatives of the loss function with respect to the weights of all the layers, and stacking up those derivatives we get $\nabla_{Loss,W}$.

5.4 Managing complexity

Performing double-precision floating-point matrix operations is quite a computationally intensive task. It is then important to find a way to keep under control the computational complexity of the algorithm that, at each training step, will compute $\nabla_{Loss,W}$.

Because of the chain rule, for any layer, we can write the expression for $\frac{\partial Loss}{\partial W_l}$ as:

$$\frac{\partial Loss}{\partial W_l} = a_{l-1}(\frac{\partial a_l}{\partial z_l}\frac{\partial Loss}{\partial a_l})^T = a_{l-1}(\frac{\partial Loss}{\partial z_l})^T$$

and at the same time, for any layer except the last one:

$$\frac{\partial Loss}{\partial a_l} = \frac{\partial z_{l+1}}{\partial a_l}\frac{\partial a_{l+1}}{\partial z_{l+1}}\frac{\partial Loss}{\partial a_{l+1}} = W_{l+1}\frac{\partial Loss}{\partial z_{l+1}}$$

This result is quite important: it means that if the layer l where to know the derivative of the loss function with respect to its output, it could quickly compute the derivative of the loss function with respect to its pre-activation vector $\frac{\partial Loss}{\partial z_l}$, multiply this partial derivative for its input vector to obtain $\frac{\partial Loss}{\partial W_l}$, store this result, multiply $\frac{\partial Loss}{\partial z_l}$ for W_l to obtain $\frac{\partial Loss}{\partial a_{l-1}}$ and pass this value to the preceding layer, which could replicate the same steps, pass $\frac{\partial Loss}{\partial a_{l-2}}$ down to layer $l-2$, and so on. As a result we obtain an algorithm that computes only once the terms $\frac{\partial Loss}{\partial z_l}$, which, depending on the value of l, are actually used to compute many derivatives.

This algorithm that propagates backwards the values of $\frac{\partial Loss}{\partial a}$ is called *Back-propagation* and it is the standard algorithm for computing $\nabla_{Loss,W}$ as it guarantees that the number of operations it will perform is linear with respect to the number of layers of the network.

The way Backpropagation works is simple: first a *forward pass* is performed, during which the network computes a prediction and all the layers store the values of a_l. Then a *backward pass* to compute $\nabla_{Loss,W}$ is performed, during which each layer passes on the value of $\frac{\partial Loss}{\partial a_l}$ to the preceding layer.

6 Final notes

We have started from defining what a *Neural Network* is and what goals it can accomplish, we have then determined the reason why neural networks are so powerful and widely used in applications, and finally we have dived into the details of how to apply gradient descent to the context of neural networks in order to train them.

Although these were the fundamental concepts that one needs to master in order to understand more advanced topics about neural networks, a lot remains uncovered.

In particular, it is important to stress out that the training process of a *NN*, as we have defined it, is a bit weak. Many algorithm exist that allow for a more dynamic, and therefore more effective, training process. Making use of such algorithms is crucially important if we want to succeed in using *NNs* to solve the machine learning problems of choice.