Machine Learning

Patrick Da Silva

February 9, 2018

Preface

I am using this document as a place to store my personal notes on the machine learning algorithms I am learning about. Feel free to send all your questions and comments to patrick1dasilva@protonmail.com.

Table of Contents

Chapter 1

Neural Networks

In this chapter, we describe a few basic neural network-based algorithms (without proofs of their accuracies, but we do explain the underlying principles).

1.1 Basic Neural Network

Consider the following diagram depicting a neural network with an input layer with two nodes, an output layer with two nodes and L hidden layers with 3 nodes, where L is a positive integer (nodes are also called **neurons**) :

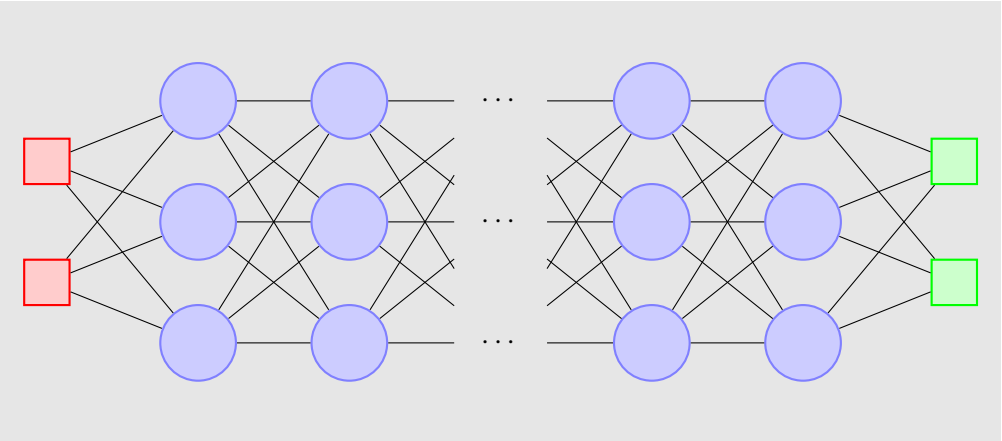

Figure 1.1: Modelization of a neural network

Input nodes displayed in red, neurons in blue and output node in green.

When the network will be trained, the idea is that input data is sent to a series of successively chained hidden layers which finally ends up to the output. Each layer consists of nodes (also commonly called "neurons", whence the name). For each layer of neurons, the data in it is sent to the next layer via linear combinations (where the coefficients, called **weights** have been determined through training with a data set) and this next layer is then "activated" by applying an activation function on each neuron. The only differences between each pair of layers are :

- the number of nodes in the preceding layer and next layer
- the coefficients of the weights to transit from the preceding layer to the next layer

- the choices of activation functions for each layer.

Note : we will abuse the chain rule. Since observations are stored as rows and linear maps are applied on those observations, for reasons explained in Subsection 1.1.3, we will write the chain rule "backwards" : if z is a function of y and y is a function of x, instead of writing

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x},$$

we will write

$$\frac{\partial z}{\partial x} = \frac{\partial y}{\partial x}\frac{\partial z}{\partial y}.$$

This is not important for scalars (i.e. functions of one-variable), but when we consider multi-dimensional calculus and the related operations, the Jacobians of our functions become matrices and the order of multiplication matters.

The reason why we do this is because of the following theorem :

Theorem 1.1.1. (Universal approximation theorem) Let φ be a non-constant, bounded and monotonically increasing continuous function and $m \geq 1$ be an integer. Given any continuous function $f : [0,1]^m \to \mathbb{R}$ and any $\varepsilon > 0$, there exists an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$ for $i = 1, \cdots, N$ such that we may define the function

$$F(x) = \sum_{i=1}^{N} v_i \varphi(w_i^\top x + b_i)$$

as an approximate realization of the function f, that is, for any $x \in [0,1]^m$, we have $|F(x) - f(x)| < \varepsilon$.

(Note for mathematicians : the latter statement is equivalent to

$$\|F - f\|_\infty = \sup_{x \in [0,1]^m} |F(x) - f(x)| < \varepsilon.$$

In other words, we can find N, w_i, v_i and b_i for $i = 1, \cdots, N$ such that F, as a function of those parameters, satisfies $\|F - f\|_\infty < \varepsilon$.)

If we think of f as the output data related to some input data that we think is a good predictor of the output, we may try to find F and use it as a model to approximate f. In practice, there are **some classes of problems**, **but not all problems**, where this function is easy to compute ; in fact, the whole idea of neural networks is based on finding this F and/or a generalization of this F.

1.1.1 Forward-propagation

Suppose that to each link between two nodes in Figure 1.1, we are given a number w_{ij} from the node i in a layer to the node j in the next layer. Those numbers will be called the **weights** of the network ; more specifically, the weight from the node i in layer ℓ to the node j in layer $\ell + 1$ will be denoted by w_{ij}^ℓ. Let's use the first layer transition as an example :

One interprets the above diagram by considering that the values in each neuron of the non-activated layer 1 (the N stands for "Non-activated") are obtained by taking linear combinations of the input, as in the following set of equations :

$$\begin{aligned} N_1^1 &= A_1^0 w_{11}^1 + A_2^0 w_{21}^1 \\ N_2^1 &= A_1^0 w_{12}^1 + A_2^0 w_{22}^1 \\ N_3^1 &= A_1^0 w_{13}^1 + A_2^0 w_{23}^1 \end{aligned}$$

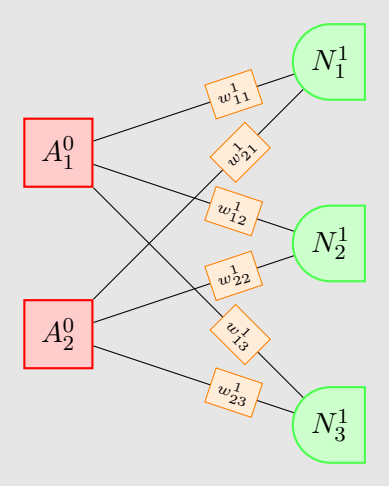

Figure 1.2: Forward-propagating from the input to layer 1

The labelling of the values A and N will be explained when we implement it in `python`.

so that each line connecting an input node i to a neuron j has weight w_{ij}^1. Recall that in data science (such as when working with R or `pandas`), when in possession of tidy data, observations are usually stored in data frames where each observation corresponds to a row and each attribute of the observation as a column. Therefore, we can re-write the above equations in matrix form :

$$\underbrace{\begin{bmatrix} N_1^1 & N_2^1 & N_3^1 \end{bmatrix}}_{\overset{\text{def}}{=} N^1} = \underbrace{\begin{bmatrix} A_1^0 & A_2^0 \end{bmatrix}}_{\overset{\text{def}}{=} A^0} \underbrace{\begin{bmatrix} w_{11}^1 & w_{12}^1 & w_{13}^1 \\ w_{21}^1 & w_{22}^1 & w_{23}^1 \end{bmatrix}}_{\overset{\text{def}}{=} W^1} \quad \Longrightarrow \quad N^1 \overset{\text{def}}{=} A^0 W^1.$$

Note : even though we display the layers vertically in the images, for a given observation in the data set, the data that will be computed in each layer is stored in row vectors.

For $i = 1, \cdots, L$ where L is the number of hidden layers, let σ_i denote the activation function of the layer i. To activate the first layer using σ_1, we simply apply the function σ_1 to each coefficient (which is the equivalent to activating each neuron), which we write as

$$\underbrace{\begin{bmatrix} A_1^1 & A_2^1 & A_3^1 \end{bmatrix}}_{\overset{\text{def}}{=} A^1} = \sigma_1(\begin{bmatrix} N_1^1 & N_2^1 & N_3^1 \end{bmatrix}) \overset{\text{def}}{=} \begin{bmatrix} \sigma_1(N_1^1) & \sigma_1(N_2^1) & \sigma_1(N_3^1) \end{bmatrix} \quad \Longrightarrow \quad A^1 \overset{\text{def}}{=} \sigma_1(N^1)$$

where we took the convention that applying a function of one variable to a matrix applies it coefficient-wise.

Note that the input layer requires no activation function, but the output layer does, so we write σ_{out} for the activation function of the output layer. In many algorithms (such as logistic neural networks, where we want to classify categorical variables instead of approximate numerical variables), we use a distinct activation function for the output layer and the other layers, whence the importance.

We can now let the first activated layer play the role of the input layer in the above diagram and repeat the process with hidden layers instead :

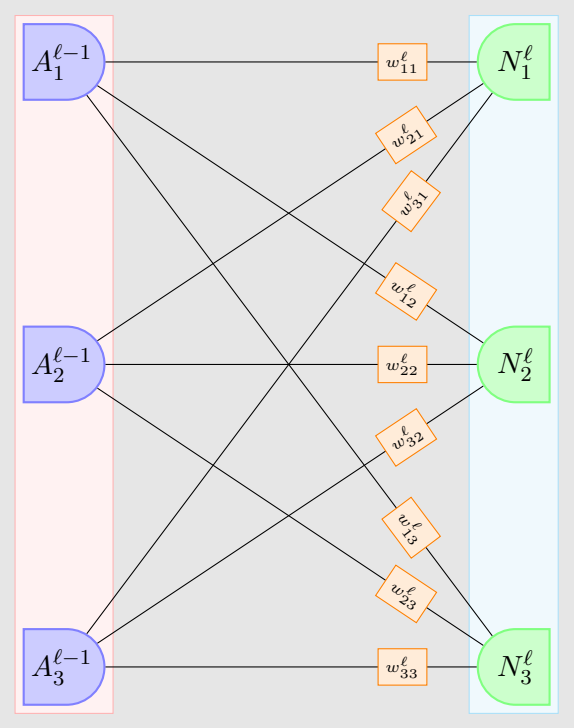

Figure 1.3: Zoom-in on the network between layers $\ell - 1$ and ℓ

Layer $\ell - 1$ is displayed in red. Layer ℓ is displayed in cyan.

which leads to the equations

$$\begin{aligned} N_1^\ell &= A_1^{\ell-1} w_{11}^\ell + A_2^{\ell-1} w_{21}^\ell + A_3^{\ell-1} w_{31}^\ell \\ N_2^\ell &= A_1^{\ell-1} w_{12}^\ell + A_2^{\ell-1} w_{22}^\ell + A_3^{\ell-1} w_{32}^\ell \\ N_3^\ell &= A_1^{\ell-1} w_{13}^\ell + A_2^{\ell-1} w_{23}^\ell + A_3^{\ell-1} w_{33}^\ell, \end{aligned}$$

or more compactly, in matrix form,

$$\underbrace{\begin{bmatrix} N_1^\ell & N_2^\ell & N_3^\ell \end{bmatrix}}_{\stackrel{\text{def}}{=} N^\ell} = \underbrace{\begin{bmatrix} A_1^{\ell-1} & A_2^{\ell-1} & A_3^{\ell-1} \end{bmatrix}}_{\stackrel{\text{def}}{=} A^{\ell-1}} \underbrace{\begin{bmatrix} w_{11}^\ell & w_{12}^\ell & w_{13}^\ell \\ w_{21}^\ell & w_{22}^\ell & w_{23}^\ell \\ w_{31}^\ell & w_{32}^\ell & w_{33}^\ell \end{bmatrix}}_{\stackrel{\text{def}}{=} W^\ell} \quad \Longrightarrow \quad N^\ell \stackrel{\text{def}}{=} A^{\ell-1} W^\ell.$$

To activate this layer, we proceed as in the case of the first layer :

$$A^\ell \stackrel{\text{def}}{=} \sigma_\ell(N^\ell).$$

We continue inductively through all the layers in this fashion until we obtain the matrix A^{out} of activated output values. Note that in our original diagram, there was a single output neuron, but nothing prevents our equations to work if there were several output values. This allows us to write some pseudocode for the following process called **forward-propagation**, when the weight matrices are given (we will see how to deal with them when we discuss back-propagation later) :

```
import numpy as np
# A list of matrices [W[0], W[1], ... , W[L][ of length L is given
# where W[l] goes from layer l to layer l+1
# A list of activation functions [activate[0], ... , activate[L]]
```

```
# where activate[l] sends N[l] to A[l+1] is given
# An input data matrix input_data is given
N = []
A = [input_data]
for l in range(0,L+1):
        N.append(np.dot(A[l], W[l]))
        A.append(activate[l](N[l]))

# The last layer in the list of vectors A is the output of our network
# Note that A has length L+2 now : the L hidden layers, input and output
# Note that because the input layer has activated nodes
# but no non-activated nodes,
# the superscripts are shifted for N compared to theory
prediction = A[-1]
```

The two fundamental equations for forward-propagation for the moment are

$$N^\ell = A^\ell W^\ell, \quad A^{\ell+1} = \sigma_\ell(N^\ell).$$

However, note that we will not use the above pseudocode explicitly since we will implement a class which will play the role of a layer. This will allows us to chain layers together without having to loop through them all the time and enable us to code both forward and back propagation with ease ; it will also remove the shift in the superscripts for N.

1.1.2 Backward-propagation

So the variable `prediction` contains the current prediction of our neural network for the ouput given the input data. This is great, but we need the neural network to actually work correctly! What we do is we gather data where the input-output pairs are known and we measure a quantity called the **cost function** or **loss function** C that we want to minimize as a function of the input and the weights. Given that function, we will use gradient descent which will allow us to tweak the weights in a way that increases the accuracy of our prediction.

Note that even though we will use backward propagation for the particular case of a cost function for numerical input and output variables, the same ideas generalize to other case scenarios.

We now fix some notation. Set

- The observations are indexed by $\alpha = 1, \cdots, M$
- $\mathrm{I} = [\mathrm{I}_0, \cdots, \mathrm{I}_{n-1}]$ to be the input vector, so that $\mathrm{I}_\alpha = [\mathrm{I}_{\alpha 0}, \cdots, \mathrm{I}_{\alpha(n-1)}]$ would be the α^{th} row of the input matrix
- $\mathrm{O} = [\mathrm{O}_0, \cdots, \mathrm{O}_{m-1}]$ to be the output vector, so that $\mathrm{O}_\alpha = [\mathrm{O}_{\alpha 0}, \cdots, \mathrm{O}_{\alpha(m-1)}]$ would be the α^{th} row of the output matrix
- $\mathrm{P} = [\mathrm{P}_0, \cdots, \mathrm{P}_{m-1}]$ to be the prediction vector, as a function of the input I and the weight matrices $W^0, \cdots, W^L$. Again, $\mathrm{P}_\alpha = [\mathrm{P}_{\alpha 0}, \cdots, \mathrm{P}_{\alpha(m-1)}]$ is the α^{th} row of the prediction matrix ; note that by definition, $\mathrm{P} = A^{L+1}$
- For each layer ℓ, the number of nodes is denoted by I_ℓ. In particular, we have $I_0 = n$ and $I_{\text{out}} = m$.

Unless otherwise mentioned, for a vector v, $\|v\|$ denotes the Euclidean norm, and for a matrix A, $\|A\|$ denotes the Frobenius norm (which is essentially the Euclidean norm when seeing the rectangular matrix A as a big vector whose coefficients are the ones of the matrix). Define

$$C(\text{input}, W^0, W^1, \cdots, W^L) = \frac{1}{2}\|\mathrm{P} - \mathrm{O}\|^2$$

To optimize our weights, we wish to compute the quantities $\frac{\partial C}{\partial w_{ij}^\ell}$ for each triplet (i, j, ℓ) so that we can adjust them via the formula

$$w_{ij}^\ell := w_{ij}^\ell - \gamma \frac{\partial C}{\partial w_{ij}^\ell}$$

where γ will be a parameter of our algorithm called the **learning rate** ; in standard gradient descent optimization, we usually perform a minimization procedure along the line passing through the point $(W^0, \cdots, W^L)$ with direction given by the gradient, but since this minimization is somewhat impossible in our situation, we take for granted that we are getting closer to our optimal solution by simply going a little bit in the direction opposite to the gradient, since this is the direction in which the cost function decreases the most, at least locally (so that small learning rates better guarantees a cost decrease).

Remark 1.1.2. Some optimizers actually allow the constant γ to vary through the various **epochs** ; an epoch is the process of computing a vector using gradients which will be used to update the weights ; for example, in a simple algorithm, the epoch would be the process of computing the partial derivatives of C with respect to the w_{ij} on the entire training data set (this is called **batch gradient descent** or BGD for short) and then applying the above formula to update the weighst. The learning rate is then a parameter which dictates how γ will vary through the epochs. We will discuss this potentially later on, but for the moment, assume γ is a constant.

For this procedure to work, we need to be able to compute the quantities $\frac{\partial C}{\partial w_{ij}^\ell}$. Fortunately, this is possible, but it requires two applications of the chain rule. We begin with the last layer (this propagation will go "backwards"!). That is, we compute the derivatives $\frac{\partial C}{\partial w_{ij}^{L+1}}$ first. Consider the following diagram :

The chain rule tells us that

$$\frac{\partial C}{\partial w_{ij}^{L+1}} = \frac{\partial N_j^{L+1}}{\partial w_{ij}^{L+1}} \frac{\partial \mathrm{P}_j}{\partial N_j^{L+1}} \frac{\partial C}{\partial \mathrm{P}_j}$$

Essentially, applying the chain rule amounts to following all the paths from w_{ij}^{L+1} to C (in this case, there is only one such path), multiplying the derivatives that we find along the way and adding up the results found on all of those paths. We can compute those three quantities. The first is related to our choice of cost function and the second to our choice of activation function, so we have

$$\frac{\partial C}{\partial \mathrm{P}_j} = \mathrm{P}_j - \mathrm{O}_j, \qquad \frac{\partial \mathrm{P}_j}{\partial N_j^{L+1}} = \sigma'_{L+1}(N_j^{L+1}).$$

As for the third, since $N^{L+1} = A^L W^{L+1}$ (e.g. $N_j^{L+1} = \sum_{i=1}^{I_{L+1}} A_i^L w_{ij}^{L+1}$), we have

$$\frac{\partial N_j^{L+1}}{\partial w_{ij}^{L+1}} = A_i^L \quad \Longrightarrow \quad \frac{\partial C}{\partial w_{ij}^{L+1}} = \left(A_i^L\right)\left(\sigma'_{L+1}(N_j^{L+1})\right)(\mathrm{P}_j - \mathrm{O}_j).$$

We move on the computation for $\frac{\partial C}{\partial w_{ij}^L}$. We use the same ideas :

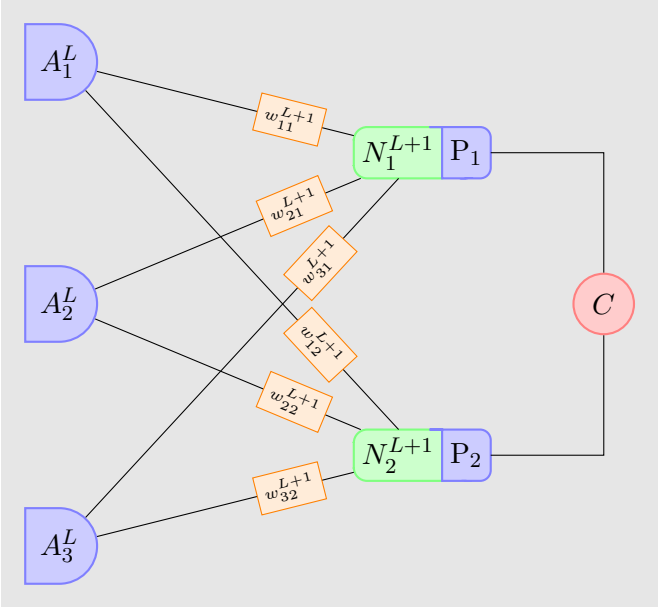

Figure 1.4: Backward-propagation on the last layer

which gives us the formula

$$\frac{\partial C}{\partial w_{ij}^L} = \frac{\partial N_j^L}{\partial w_{ij}^L}\frac{\partial A_j^L}{\partial N_j^L}\frac{\partial C}{\partial A_j^L} = \left(A_i^{L-1}\right)\left(\sigma_L'\left(N_j^L\right)\right)\left(\frac{\partial C}{\partial A_j^L}\right).$$

Even though one would be tempted to write

$$\frac{\partial C}{\partial A_j^L} = \sum_{\substack{k \text{ in} \\ \text{output layer}}} \frac{\partial w_{jk}^{L+1}}{\partial A_k^L}\frac{\partial C}{\partial w_{jk}^{L+1}}$$

(by going back along all the paths from A_j^L to the node corresponding to C and use the fact that we already know $\frac{\partial C}{\partial w_{jk}^L}$ from our previous computations), this last partial derivative makes no sense since A_j^L is a function of $W^0, \cdots, W^L$ and W^{L+1} is independent of those variables. Therefore, we need a different approach. Instead, we use the chain rule in the same way that we did for the partial derivatives of C with respect to w_{jk}^{L+1}, but this time starting from the node corresponding to A_j^L and "overlooking" w_{ij}^{L+1}. By looking at Figure 1.4, we deduce

$$\frac{\partial C}{\partial A_j^L} = \sum_{\substack{k \text{ in} \\ \text{output layer}}} \frac{\partial N_k^{L+1}}{\partial A_j^L}\frac{\partial \mathrm{P}_k}{\partial N_k^{L+1}}\frac{\partial C}{\partial \mathrm{P}_k} = \sum_{\substack{k \text{ in} \\ \text{output layer}}} \left(w_{jk}^{L+1}\right)\left(\sigma_{L+1}'(N_k^{L+1})\right)(\mathrm{P}_k - \mathrm{O}_k),$$

which implies that we know the quantities $\frac{\partial C}{\partial w_{ij}^L}$.

Repeating those ideas inductively (one can get a picture by replacing L by ℓ in Figure 1.5), we obtain the formulas

$$\forall \ell = 1, \cdots, L+1, \qquad \frac{\partial C}{\partial w_{ij}^\ell} = \frac{\partial N_j^\ell}{\partial w_{ij}^\ell}\frac{\partial A_j^\ell}{\partial N_j^\ell}\frac{\partial C}{\partial A_j^\ell} = A_i^{\ell-1}\left(\sigma_\ell'(N_j^\ell)\right)\left(\frac{\partial C}{\partial A_j^\ell}\right)$$

$$\forall \ell = 2, \cdots, L+1, \qquad \frac{\partial C}{\partial A_j^{\ell-1}} = \sum_{k \text{ in layer } \ell} \frac{\partial N_k^\ell}{\partial A_j^{\ell-1}}\frac{\partial A_k^\ell}{\partial N_k^\ell}\frac{\partial C}{\partial A_k^\ell} = \sum_{k \text{ in layer } \ell} \left(w_{jk}^\ell\right)\left(\sigma_\ell'\left(N_k^\ell\right)\right)\left(\frac{\partial C}{\partial A_k^\ell}\right).$$

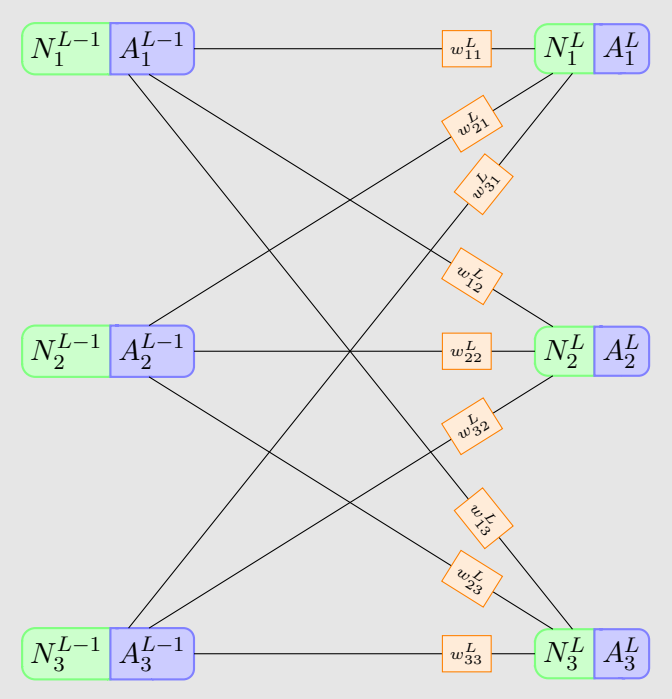

Figure 1.5: Backward-propagation on the forelast layer

Note that those formulas agree with the case of the last hidden layer if we replace $\frac{\partial C}{\partial A_k^\ell}$ by

$$\frac{\partial C}{\partial A_k^{\text{out}}} = \frac{\partial C}{\partial \mathrm{P}_k} = \mathrm{P}_k - \mathrm{O}_k,$$

which means that there should be no difference in implementation.

Remark 1.1.3. One recalls from calculus that if we have a function $f : \mathbb{R}^n \to \mathbb{R}$ where the vector in the domain $\mathbb{R}^n$ is written in column form, the gradient ∇f is written in row form as to emphasize the fact that the gradient belongs to the dual of the tangent space of $\mathbb{R}^n$; since we are working with row vectors, this should mean that the gradient has to be a column vector. This mathematical artifact is not important for us ; what is more important is efficient computation. Since our "vectors" (i.e. the data computed in the layers) are denoted as matrices (where each row corresponds to the values in the neurons of a layer corresponding to one input row, namely one input data observation unit), it will be more practical to store the values of $\frac{\partial C}{\partial A_k^\ell}$ and $\frac{\partial C}{\partial w_{ij}^\ell}$ in matrices of the same format as A^ℓ and W^ℓ, respectively.

1.1.3 Implementation

The only question that remains is how to implement those in terms of `numPy` operations, since this is how our data is stored in `python`. In the above, we know how to forward-propagate and backward-propagate a single observation of data, which is rarely what we want in practice ; in practice, we are not given an input vector $A^0 = [A_0^0, \cdots, A_{n-1}^0]$, but rather an input matrix

$$A^0 = \begin{bmatrix} A_{00}^0 & \cdots & A_{0(n-1)}^0 \\ A_{10}^0 & \cdots & A_{1(n-1)}^0 \\ \vdots & \cdots & \vdots \\ A_{(M-2)0}^0 & \cdots & A_{(M-2)(n-1)}^0 \\ A_{(M-1)0}^0 & \cdots & A_{(M-1)(n-1)}^0 \end{bmatrix}$$

where each row vector $[A^0_{\alpha 0}, \cdots, A^0_{\alpha(n-1)}]$ corresponds to an observation. One important note : in classical linear algebra, vectors are considered as column vectors and matrices as linear maps via multiplication on the left. Since we are dealing with row vectors, applying a linear map to a row vector corresponds to multiplication on the **right**, not on the left. More importantly, the chain rule also reads backwards since the composition of linear maps (the Jacobians of our composed functions) corresponds to matrix multiplications, all on the right.

Forward-propagation

This one is quite straight-forward : each layer ℓ corresponds to a matrix of non-activated values $N^\ell = [N^\ell_{\alpha j}]_{\alpha,j}$ and activated values $A^\ell = [A^\ell_{\alpha j}]_{\alpha,j}$ where the rows correspond to each observation and the columns to each neuron in the layer. Then, recalling our convention that applying a numerical function to a matrix means applying it coefficient-wise, we have the formulas

$$\text{for } \ell = 0, \cdots, L, \qquad N^\ell(A^\ell, W^\ell) = A^\ell W^\ell, \qquad A^{\ell+1}(N^\ell) = \sigma_\ell(N^\ell)$$

$$\mathrm{P} = A^{L+1}.$$

The functions we will use in `numPy` already enable coefficient-wise application of functions, so this will not give us too much trouble.

Backward-propagation

If one tries to picture what happens to the dimensions of the matrices (and tensors!) involved using boxes for representing row vectors and matrices, in a picture where each input A^0_α is a row and each W^L is a matrix, the dimension corresponding to α (the subscript which ranges over observation units) is different from that relating rows and columns of W ; for that reason, it makes more sense to picture it as "depth", and is therefore always the "last" dimension when ordering dimensions. In other words, when performing our operations, we are working with tensors of order 3, where the dimensions of those tensors corresponds to the dimension of the preceding layer, the dimension of the next layer, and the number of observations in the data set. The only tensors whose coefficients do not vary across observations in the data set are the tensors corresponding to the weight matrices (indeed, we do not want our weights to be functions of the input!).

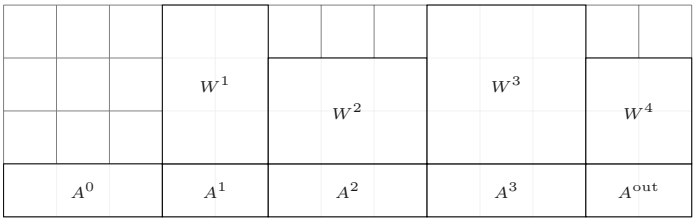

Figure 1.6: Dimension analysis of the matrices A^ℓ and W^ℓ

In this particular example, we have three hidden layers. The input is 3-dimensional and the output 2-dimensional. The position of the row vectors A^ℓ and the weight matrices W^ℓ should remind the reader of the position to put those matrices into to perform matrix multiplication by hand. The dimensions, i.e. the number of neurons/nodes in each layer, are equal to $(I_0, I_1, I_2, I_3, I_{\text{out}}) = (3, 2, 3, 3, 2)$ in this example. One can use this picture to see how the parameter α corresponds to the dimension of depth, not width or height. To multiply those tensors, just perform matrix multiplication on "each α-slice". Warning : there is no pre-defined method to multiply two tensors of order 3.

As in our inspection of dimensions explained above, set

$$\forall \ell = 1, \cdots, L, L+1, \qquad \frac{\partial C}{\partial A^\ell} \overset{\text{def}}{=} \left[\frac{\partial C_\alpha}{A^\ell_{\alpha j}}\right]_{\alpha,j} \qquad \frac{\partial C}{\partial W^\ell} \overset{\text{def}}{=} \left[\frac{\partial C_\alpha}{W^\ell_{ij}}\right]_{\alpha,i,j}$$

Note that $\frac{\partial C}{\partial W^\ell}$ is a 3-dimensional array, i.e. a tensor of order 3. When we fix α (i.e. an observation), the corresponding values in $\frac{\partial C_\alpha}{W^L_{ij}}$ gives us the gradient of C_α, the cost function computed on the α^{th} observation.

Unlike ordinary matrices where there is only one way to define the product of two matrices A, B (as AB if the number of columns of A equals the number of rows of B), there is no clear way to multiply two tensors of order 3 ; suppose A and B were two "cubic tensors" (the corresponding box of numbers is a perfect cube, i.e. all sides have the same length) ; we could do matrix multiplication along "slices", but then along which pairs of sides do we perform the multiplications?

To remedy to this problem, we will explicitly define an operation which will suit our needs and indicate later how to implement it in `numPy`. Speaking of operations, another one which will be useful is the **Hadamard product** of two matrices of the same shape : if $A = (a_{ij})_{i,j}, B = (b_{ij})_{i,j}$ are two $m \times n$ matrices, then their Hadamard product is defined by coefficient-wise multiplication :

$$(A \odot B)_{i,j} \overset{\text{def}}{=} (a_{ij} b_{ij})_{i,j}.$$

It is an easy exercise to see that the formulas for backward-propagation are given by the formulas

$$\frac{\partial C}{\partial A^{\ell-1}} = \left[\frac{\partial C_\alpha}{A^{\ell-1}_{\alpha j}}\right]_{\alpha,j} = \underbrace{\left(\sigma'_\ell(N^\ell) \odot \frac{\partial C}{\partial A^\ell}\right)}_{(\alpha,k)} \underbrace{(W^\ell)^\top}_{(k,j)}$$

$$\frac{\partial C}{\partial W^\ell} = \left[\frac{\partial C_\alpha}{\partial W^\ell_{ij}}\right]_{\alpha,i,j} = \underbrace{A^{\ell-1}}_{(\alpha,i)} \boxtimes \underbrace{\left(\sigma'_\ell(N^\ell) \odot \frac{\partial C}{\partial A^\ell}\right)}_{(\alpha,j)}$$

(the order of the subscripts of the matrices are written under the curly braces) where the operation $\boxtimes$ is defined as follows : if a, b, c are integers, A is a $a \times b$ matrix and B is a $a \times c$ matrix, then

$$A \boxtimes B = (a_{\alpha i})_{\alpha,i} \boxtimes (b_{\alpha j})_{\alpha,j} \overset{\text{def}}{=} (a_{\alpha i} b_{\alpha j})_{\alpha,i,j}.$$

This is the equivalent of considering for each α_0 the column vector $A_{\alpha 0}$, the row vector B_{α_0}, taking their product $A_{\alpha_0} B_{\alpha_0}$ as matrices and storing the resulting matrix as the α_0^{th} "slice" of the tensor $A \boxtimes B$ of order 3 with shape (a, b, c).

Remark 1.1.4. It is not very surprising to see that the transpose matrix $W^\top$ appears in the formula ; since multiplying by W^ℓ goes from layer $\ell - 1$ to the layer ℓ, multiplying by $(W^\ell)^\top$ goes from layer ℓ to layer $\ell - 1$; this makes sense since we are doing back-propagation, i.e. going backwards.

Gradient descent : batch gradient descent (BGD) and stochastic gradient descent (SGD)

Now that we have all the data we need (i.e all the partial derivatives concerning all of our data), we can perform gradient descent. Unlike the standard gradient descent algorithm where the step size γ is allowed to change after each iteration, we fix this step size to simplify the computations ; this is called the **learning rate** of our algorithm. We have two options presented to us :

- *Batch Gradient Descent (BGD).* With this technique, we simply perform all the operations above on all of our training data and take the average gradient contributed by each data input. This gives

$$W_{ij}^{\ell} \coloneqq W_{ij}^{\ell} - \gamma \left(\frac{1}{M} \sum_{\alpha=1}^{M} \frac{\partial C_{\alpha}}{\partial W_{ij}^{\ell}} \right).$$

  This has the inconvenience of being computationally expensive since we have to go through our entire training set.

- *Stochastic Gradient Descent (SGD).* We sample one observation α from our training data set and compute the gradient with this input. This gives

$$\alpha \text{ sampled from } \{1, \cdots, M\}$$
$$W_{ij}^{\ell} \coloneqq W_{ij}^{\ell} - \gamma \frac{\partial C_{\alpha}}{\partial W_{ij}^{\ell}}.$$

  This has the advantage that an epoch computes extremely quickly, but of course, this algorithm is a bit more "wiggly" in the sense that instead of using a very deterministic path from our initial weights to the optimal weights as in BGD, we take a stochastic path, i.e. a random walk in the space of all weights. This makes the theory a bit harder but the application a lot simpler.

- *Mini-batch Gradient Descent (MBGD).* Instead of using all of our data in one iteration, or just one observation as in stochastic gradient descent, we take a small sample of our data at each iteration and update the weights using those samples. This is somewhere between BGD and SGD as both are degenerate cases of this technique. The formulas are :

$$\alpha_1, \cdots, \alpha_r \text{ sampled from } \{1, \cdots, M\}$$
$$W_{ij}^{\ell} \coloneqq W_{ij}^{\ell} - \gamma \left(\frac{1}{r} \sum_{s=1}^{r} \frac{\partial C_{\alpha_s}}{\partial W_{ij}^{\ell}} \right).$$

  This gives us two parameters, the learning rate γ and the sample size r. (We will actually use a "sample proportion", i.e. what percentage of our data set will be used for each batch ; this has the same effect but is easier to visualize.)

Remark 1.1.5. When performing MBGD over large training data sets, there is not much difference between sampling with or without repetition. However, since we only want our algorithm to work, it is not really an important factor ; choose whatever suits you. In our case, we will sample using the `random` library which will do the work for us.

Remark 1.1.6. The role of α as a subscript will be somewhat invisible in the code since we will average over all observations treated in one back-propagation pass using commands such as `np.einsum`, so that we do not need to actually save the values of the derivatives for each observation, but rather the average. This saves a lot of memory and makes training much more efficient.

Code using numPy

Note : in this entire subsection, `numpy` has been imported as `np`. We'll also omit the lines of code which raise errors and the such ; those are freely available in the code examples provided with this document.

Some remarks before we begin :

- In an ideal theoretical world, we would store all our data in 3-dimensional arrays so that the picture of Figure 1.6 fits the code, but in practice, this is inefficient. Only the "really" 3-dimensional array (i.e. has potentially dimension greater than 1 in all directions) will be treated as such, namely the partial derivatives of C with respect to the coefficients of W^ℓ.

- Our neural network and the layers of the network will be implemented as classes. Each layer will exchange data with its preceding layer via its forward-propagate and back-propagate methods. The training method will apply the backward-propagation method repeatedly and update the weights via the gradients computed by back-propagating.

- To uniformize the computations, we will rescale the input and output as numerical variables between 0 and 1 using a variant of the following linear "rescale" function :

$$x \mapsto \frac{x - \min(x)}{\max(x) - \min(x)}$$

  which has the advantage of being easily invertible, with the following inverse "unscale" map :

$$y \mapsto (\max(y) - \min(y))y + \min(y).$$

  (The variant just fixes $\min(x)$ and $\max(x)$ in advance as "scales" instead of computing them.) The rescaling allows us to keep a uniform standard when data goes through our algorithm ; otherwise, we would have to rescale all the other operations of our network to adapt to our data, which is undesirable (explicitly, we would have to rescale the activation functions and their derivatives ; this is not important when using ReLU since it is invariant under scaling, but it becomes very important for functions such as the sigmoid or the softmax for classifiers, one of the main uses of neural networks).

- We will need activation functions to be ready to use with `numPy`. Here are the three most common examples we will use :

  - The sigmoid :

```
def sigmoid(z):
    return 1/(1+np.exp(-z))

def sigmoid_prime(z):
    # We exploit the fact that the derivative of the sigmoid satisfies
    # sigmoid' = sigmoid * (1-sigmoid)
    # to avoid computing two exponentials and potentially accelerate the algorithm
    x = sigmoid(z)
    return x*(1-x)
```

  - The Rectified Linear Unit, also known as ReLU :

```
def relu(z):
    return np.where(z < 0, 0, z)

def relu_prime(z):
    return np.where(z < 0, 0, 1)
```

  - The Leaky Rectified Linear Unit, also known as Leaky ReLU or LReLU :

```
def leakyrelu(z, leak = 0.01): # Leak = 0 makes this the same as ReLU
    return np.where(z < 0, leak*z, z)

def leakyrelu_prime(z,leak = 0.01):
    return np.where(z < 0, leak, 1)
```

We being by implementing a `Layer` class, which will be used to construct our neural network :

```
class Layer(object):
    def __init__(self, number_of_neurons,
                       activation_function = None,
                       preceding_layer = None): # Hidden layers are those for which this is not None
        self.number_of_neurons = number_of_neurons
        self.preceding_layer = None
        self.activate_layer = None
        self.activate_layer_prime = None

        if preceding_layer is not None: # Or in other words, if this is not the input layer
            self.preceding_layer = preceding_layer
            self.weights = np.random.randn(self.preceding_layer.number_of_neurons,
                                           self.number_of_neurons)
            self.weights_gradient = np.zeros([self.preceding_layer.number_of_neurons,
                                              self.number_of_neurons])
            self.A_gradient = np.zeros(self.number_of_neurons)

            if activation_function == "relu":
                self.activate_layer = relu
                self.activate_layer_prime = relu_prime
            elif activation_function == "sigmoid":
                self.activate_layer = sigmoid
                self.activate_layer_prime = sigmoid_prime
            elif activation_function == "leakyrelu":
                self.activate_layer = leakyrelu
                self.activate_layer_prime = leakyrelu_prime
```

Since our neural network model just chains up one layer after the other, all we need is to construct this chain. When implementing our neural network class, we will begin with the input layer and successively construct the next layers by indicating in a layer we just created what is the layer coming before it, which is necessary to know the dimensions of its weight matrix. The attributes `self.weights_gradient` and `self.A_gradient` correspond to the matrices $\left(\frac{\partial C}{\partial w_{ij}^{\ell}}\right)_{ij}$ and $\frac{\partial C}{\partial A_i^{\ell}}$ we constructed earlier respectively, but `self.weights_gradient` actually only contains one gradient in the strict sense if we are using the SGD optimizer ; when using BGD or MBGD, it will contain the average of the gradients over our batch.

Note that the weights are initialized randomly for the simple reason that we have no idea in practice how to initialize them. Either way, we are hoping that the gradient descent algorithm will converge to an appropriate collection of weights, so that the choice of initial guess is not entirely important. It seems important in practice, however, not to choose them too big or too small ; a normal distribution with mean 0 and variance 1 usually does the trick.

This `Layer` class comes together with two methods, one to forward-propagate and one to update the weights :

```
    def forward_propagation(self, input_data = None):
        if self.preceding_layer is not None:
            return self.activation_function(
                               np.dot( self.preceding_layer.forward_propagation(input_data),
                                       self.weights ))
        else: # If we hit this, we are dealing with the input layer
            return input_data
```

```
    def update_weights(self, learning_rate):
        self.weights -= learning_rate * self.weights_gradient
        self.weights_gradient = np.zeros([self.preceding_layer.number_of_neurons,
                                          self.number_of_neurons])
```

We will not need a back-propagation method ; this will be taken care of by the optimizer, which is easier this way since the forward pass needed in the optimizer also needs to store all the intermediate values

obtained during the forward pass to enable computing derivatives, something which we will do on the fly.

We can now define the `Neural_Network` class :

```
class Neural_Network(object):
    def __init__(self, input_scales, hidden_layers_sizes, hidden_layers_activation_functions,
                       output_scales, output_layer_activation_function="relu"):
        self.input_scales = input_scales
        self.output_scales = output_scales
        input_layer_size = len(input_scales)
        output_layer_size = len(output_scales)

        self.layers_list = [Layer(input_layer_size)]
        if len(hidden_layers_sizes) != len(hidden_layers_activation_functions):
            raise ValueError("The list of layers sizes must have the
                              same length as the list of layers' activation functions")

        # Building the computation graph (it is a line graph in this case)
        for i in range(len(hidden_layers_sizes)):
            self.layers_list.append(Layer(hidden_layers_sizes[i],
                                          hidden_layers_activation_functions[i],
                                          self.layers_list[-1]))

        self.layers_list.append(Layer(output_layer_size,
                                      output_layer_activation_function,
                                      self.layers_list[-1]))
```

It comes with four methods :

- The `rescale_data` method, which rescales the data according to the scales the data has
- The `run_network` method, which makes a forward pass to obtain a prediction based on input value
- The `gradient_descent_optimizer`, which optimizes the weights given a batch of data (whether it is a single observation or a batch, it runs equally well)
- The `train_network` method, which applies the optimizer method and keeps track of the metrics such as cost and accuracy.

We explain those methods in order, beginning with `rescale_data` :

```
def rescale_data(self, data, bounds_array, in_reverse=False):
    if in_reverse:
        return (bounds_array.T[1] - bounds_array.T[0])*data + bounds_array.T[0]
    else:
        return (data - bounds_array.T[0])/(bounds_array.T[1] - bounds_array.T[0])
```

It is a good exercise of vectorization to read through this algorithm and understand why this definition of `rescale_data` is blazingly fast ; we are exploiting the power of `numPy` matrix operations to speed up the rescaling process. A good start would be to run `IPython` in a terminal and test those functions on some example matrices and scales. Given an input `bounds_array` which corresponds to a $n \times 2$ matrix

$$[[l_1, u_1], \cdots, [l_n, u_n]]$$

of pairs $[l_i, u_i]$ where l_i is the lower bound of the i^{th} feature (i.e. column) of the input `data` matrix (and u_i the upper bound), we rescale column i according to the scale $[l_i, u_i]$. Onto the `run_network` method :

```
def run_network(self, input_data):
    return self.rescale_data(
        self.layers_list[-1].forward_propagation(self.rescale_data(input_data, self.input_scales)),
        self.output_scales,
        in_reverse=True)
```

In other words, we rescale the data so that the inputs land between 0 and 1, make a forward pass, and unscale the data to the scales of the output.

Now the optimizer. Note that this method is optimizing only for one batch of input data, i.e. the method is used to back-propagate **once**. The `train_network` method will then do several calls to the optimizer to obtain the trained weights.

```
def gradient_descent_optimizer(self, input_data, output_data, learning_rate):
    training_data = self.rescale_data(input_data, self.input_scales)
    number_of_inputs = input_data.shape[0]
    A = [training_data]
    N = [None] # Just to shift the indices and make them match with theory
    for layer in self.layers_list[1:]: # We already have the input layer's values
        N.append(np.dot(A[-1], layer.weights))
        A.append(layer.activate_layer(N[-1]))

    self.layers_list[-1].A_gradient = A[-1] - self.rescale_data(output_data, self.output_scales)
    for l in reversed(range(1,len(self.layers_list))):
        layer = self.layers_list[l]
        layer.weights_gradient = np.einsum(
            "ai,aj->ij",
            A[l-1],
            layer.activate_layer_prime(N[l]) * layer.A_gradient / number_of_inputs)
        layer.update_weights(learning_rate)

        if l > 1:
            layer.preceding_layer.A_gradient = np.dot(
                layer.activate_layer_prime(N[l]) * layer.A_gradient ,
                layer.weights.T )
```

We begin by "manually" (i.e. without the `forward_propagation` method of the layers) computing the vectors N_α^ℓ and A_α^ℓ for each observation $\alpha \in \{1, \cdots, M\}$, which is done via the matrix algorithms of `numPy`, and then store those in lists conveniently named N and A. After that, we use the data of those lists to start updating the values of `self.weights_gradient` of each layer, and finally update the weights of each layer using its `update_weights` method. Note that the `einsum` function does exactly what one expects from the operation $\boxtimes$ if we replace `''ai,aj->ij''` by `''ai,aj->aij''`, but since we are going to compute an average gradient over all gradients we are getting from the input data, we are removing that `a` from `''ai,aj->aij''`, which in Einstein summation notation has the effect of replacing a tensor of order 3 of the form $(x_{\alpha ij})_{\alpha ij}$ with the tensor of order 2 (i.e. the matrix)

$$\left(\sum_{\alpha=1}^{M} x_{\alpha ij} \right)_{ij}$$

In other words, it performs the operation $\boxtimes$ and then sums the result over all observations.

Finally, it remains to look at the `train_network` method, which trains it via an MSGD algorithm :

```
def train_network(self, input_data, output_data, number_of_epochs, display_performance_frequency,
        training_split, validation_split, test_split, batch_size_split = 1, learning_rate = 0.01,
        data_accuracy=None):
    number_of_observations = len(input_data)
    split_row_indices = np.dot( np.random.multinomial(1,
                                                      [training_split, validation_split, test_split],
                                                      number_of_observations),
                                np.array([[0],[1],[2]]) ).flatten()

    training_input_data = input_data[split_row_indices == 0]
    training_output_data = output_data[split_row_indices == 0]
    validation_input_data = input_data[split_row_indices == 1]
    validation_output_data = output_data[split_row_indices == 1]
    testing_data = input_data[split_row_indices == 2]
```

```
    training_data_size = len(training_input_data)
    validation_data_size = len(validation_input_data)
    sample_size = int(batch_size_split * training_data_size)
    i = 1
    training_cost = []
    validation_cost = []
    accuracy = []
    for i in range(number_of_epochs):
        sampling_row_indices = np.random.randint(training_data_size, size=sample_size)
        input_batch = training_input_data[sampling_row_indices,:]
        output_batch = training_output_data[sampling_row_indices,:]

        self.gradient_descent_optimizer(input_batch, output_batch, learning_rate)
        training_cost.append( norm_squared(self.run_network(training_input_data)
                                           - training_output_data)
                              / training_split )
        validation_cost.append( norm_squared(self.run_network(validation_input_data)
                                           - validation_output_data)
                              / validation_split )
        accuracy.append( 1 - norm_squared( threshold( self.run_network(validation_input_data), 0.5 )
                                           - validation_output_data )
                         / validation_data_size )

        if i \% display_performance_frequency == 0:
            print("Epoch : ", i, "/", number_of_epochs)
            print("Cost function on training set : ", training_cost[-1])
            print("Cost function on validation set : ", validation_cost[-1])
            print("Accuracy on validation set : ", accuracy[-1])
            if data_accuracy is not None:
                print("Data accuracy : ", data_accuracy)
            print("")

    return training_cost, validation_cost, accuracy # This is for plotting purposes
```

(Note : I have included a splitting of the data to also include a data set for testing, but I have not implemented anything involving the test data set yet. So far, I only did tests by repeatedly using the validation data since anyway each run of my program generated random data, as we will see in the examples. In practice, this is not possible! One does not have an endless supply of data, so that's why we keep a proportion of it for testing. Implementing testing could make this code useful in the future.)

We begin by defining a vector of length M (`split_row_indices`) whose component α will be 0 if it belongs to the training set, 1 if it belongs to the validation set and 2 if it belongs to the testing set. We then use those indices to form the data sets for training, validation and testing.

A list of training and validation costs is stored for plotting purposes, as well as one for accuracy. For each epoch, we perform sampling within the data set of $100*$`batch_size_split` percent of the training data to use for training. We then run the optimizer on this batch and record the metrics which are finally printed. The lists of metrics are returned to enable sending them to some plotting library of your choice when applying this class to your data.

Also note that our definition of accuracy in this method is specific to the case of **binary classifiers**, as we will see in the examples. If we were to classify data among multiple options, it would probably have to look different, but since our examples will be binary, let us keep it as it is for now.

1.1.4 Adding bias

In a neural network, the **bias** of a neuron is essentially the b in the substitution of the map $N^{\ell+1}(A^\ell, W^{\ell+1}) = A^\ell W^{\ell+1}$ by the map

$$N^{\ell+1}(A^\ell, W^{\ell+1}, B^{\ell+1}) = A^\ell W^{\ell+1} + B^{\ell+1}.$$

Using this has many advantages, one of them being that it interprets the Universal approximation theorem correctly, for starters. If we think of a neural network as a bunch of neurons firing electricity as in a brain, one can think of the bias as a flux of eletricity which fires anyway regardless of input, and the weights diminish or boost that firing strength.

On a practical level, it is very useful to re-write the above as follows : writing $m \overset{\text{def}}{=} I_\ell$ and $n \overset{\text{def}}{=} I_{\ell+1}$ for simplicity, we obtain

$$AW + B = \begin{bmatrix} a_1 & \cdots & a_m \end{bmatrix} \begin{bmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{m1} & \cdots & w_{mn} \end{bmatrix} + \begin{bmatrix} b_1 & \cdots & b_n \end{bmatrix} = \underbrace{\begin{bmatrix} a_1 & \cdots & a_m & 1 \end{bmatrix}}_{\overset{\text{def}}{=} \widetilde{A}^\ell} \underbrace{\begin{bmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{m1} & \cdots & w_{mn} \\ b_1 & \cdots & b_n \end{bmatrix}}_{\overset{\text{def}}{=} \widetilde{W}^{\ell+1}}$$

The beauty of this reformulation $(A^\ell, W^{\ell+1}, B^{\ell+1}) \mapsto (\widetilde{A}^\ell, \widetilde{W}^{\ell+1})$ is that it leaves our formulas for back-propagation almost completely unchanged! Take a look :

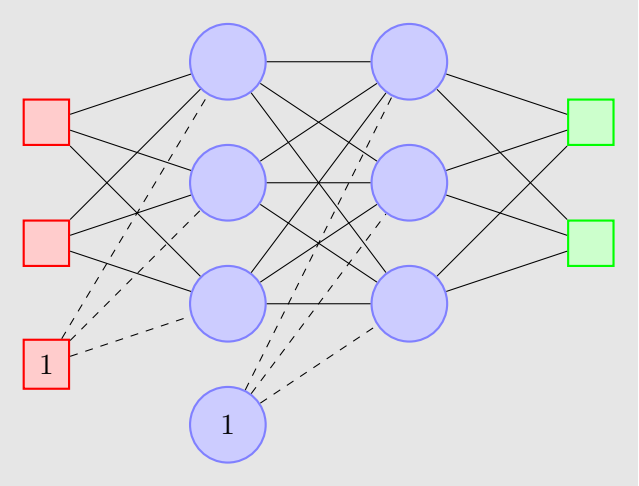

Figure 1.7: Modelization of a neural network with bias

Input nodes displayed in red, neurons in blue and output node in green. Bias nodes are marked with a 1. It has two input nodes and two output nodes with two hidden layers with three nodes each ; the input layer and first hidden layer have bias, the next two don't.

The formulas for forward-propagation change slightly, but not in a very complicated manner ; all we have to do is adjoint a 1 to A and add a row to W ; the values $b_1, \cdots, b_n$ can then be thought of as "weights for the constant node" which will measure the bias. This 1 is added in the formation of the matrix A when forward-propagating, i.e. we append 1 to the vector $\sigma_\ell(N^\ell)$.

To obtain the formulas for back-propagation, we need the quantities $\frac{\partial C}{\partial \widetilde{w}_{ij}^\ell}$ where $\widetilde{w}_{ij}^\ell$ is just the $(i,j)^{\text{th}}$ coefficient of $\widetilde{W}^\ell$. Note that we should not compute $\frac{\partial C}{\partial \widetilde{A}^{\ell-1}}$; the node corresponding to the bias should

be seen as a constant node, and if it doesn't vary, I cannot compute a partial derivative of C with respect to that node. The reasoning for those formulas is the same, except that we follow paths along nodes in a diagram such as in Figure 1.7 :

$$\frac{\partial C}{\partial A^{\ell-1}} = \left[\frac{\partial C_\alpha}{A^{\ell-1}_{\alpha j}}\right]_{\alpha,j} = \underbrace{\left(\sigma'_\ell(N^\ell) \odot \frac{\partial C}{\partial A^\ell}\right)}_{(\alpha,k)} \underbrace{(W^\ell)^\top}_{(k,j)}$$

$$\frac{\partial C}{\partial \widetilde{W}^\ell} = \left[\frac{\partial C_\alpha}{\partial \widetilde{W}^\ell_{ij}}\right]_{\alpha,i,j} = \underbrace{\widetilde{A}^{\ell-1}}_{(\alpha,i)} \boxtimes \underbrace{\left(\sigma'_\ell(N^\ell) \odot \frac{\partial C}{\partial A^\ell}\right)}_{(\alpha,j)}$$

The reason why the back-propagation formulas remain almost unaffected is because the partial derivative $\frac{\partial C}{\partial b_j}$ is determined by the same formulas as $\frac{\partial C}{\partial w_{ij}}$, except that $\frac{\partial N^\ell_j}{\partial b_j}$ equals 1, which turns out to be the 1 we adjoined to the matrix $A^{\ell-1}$. This tells us to replace $A^{\ell-1}$ by $\widetilde{A}^{\ell-1}$, but we still need to use W for the $(W^\ell)^\top$ term since we don't add a 1 to the vector $\frac{\partial C}{\partial A^{\ell-1}}$.

1.1.5 Examples of binary classifiers

Recall the discussion done at the beginning of Section 1.1 concerning the Universal approximation theorem. If we want to build a neural network to approximate a certain function, our job is to brilliantly "guess" the function F which models our data given by the function f (which we never know completely). Note that one should think of our data also as the function f plus some random noise not explained by the input data, which is why it makes sense to use the data anyway to try to approximate f, which then gives us a good prediction of what the output data should be given the input. To do those guesses, we will often make assumptions on the so-called **hyperparameters**, i.e. those parameters of the neural network which we do not obtain by training but by manually inputting them. Examples of hyperparameters include the learning rate, the number of hidden layers, the number of nodes per layers, the training/validation/testing ratios, etc.

A very good class of problems on which neural networks perform beautifully is on **classifiers**, i.e. input data which predicts **categorical data** : given input data, is this a cat or a dog, is this red or blue, what digit from 0 to 9 is displayed, etc. When there are only two possible outputs (such as "cat" or "not cat"), we speak of a **binary classification problem** and the neural network is called a **binary classifier model**. We discuss two examples below.

Logistic regression

Logistic regression is a very simple idea for binary classification problems : we are given two finite sets of n-dimensional numerical inputs (i.e. each input is a vector $x = (x_1, \cdots, x_m) \in \mathbb{R}^m$), i.e. our input data sets. One outputs the first category (we will call it category 0, colored in red) and the other one outputs the other category (which we will call 1, colored in blue).

Definition 1.1.7. Two input data sets as described above are called **linearly separable** if there exists an affine function $\omega : \mathbb{R}^m \to \mathbb{R}$, i.e. $\omega(x_1, \cdots, x_m) = x_1 w_1 + \cdots + x_m w_m + b = x^\top w + b$ where $w = (w_1, \cdots, w_m)$, such that $\omega(x) > 0$ when x is in the first data set and $\omega(x) < 0$ when x is in the second data set.

Remark 1.1.8. Affine functions are very often incorrectly called linear functions, but linear functions have to satisfy $\omega(0) = 0$. An affine function is the sum of a linear function and a constant function, i.e. $\omega(x) = x^\top w + b$ is affine because $x \mapsto x^\top w$ is linear in x.

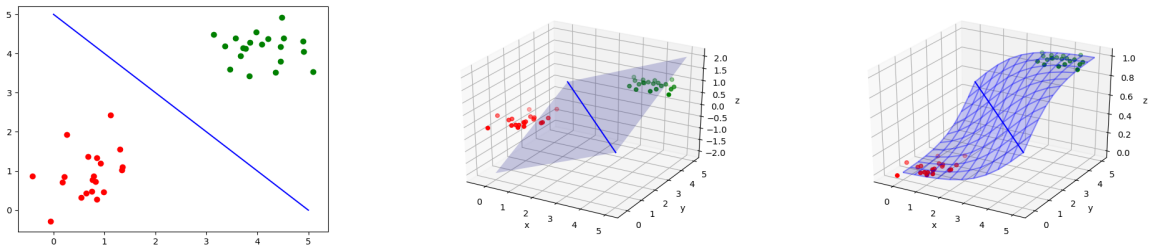

Figure 1.8: A linearly separable pair of sets of data

Input data points from category 0 are displayed in red, those from category 1 in green. The existence of the affine function ω in the definition, whose graph is depicted in the 3D plot in the middle, corresponds to the existence of the blue line on the plot on the left ; the affine function gives us the line as the set of points where $\omega(x_1, x_2) = 0$. The green points with coordinates (x_1, x_2) all satisfy $\omega(x_1, x_2) > 0$, and the red ones $\omega(x_1, x_2) < 0$. The surface depicted on the right is an example of a logistic regression model of the form $(x_1, x_2) \mapsto \sigma(\omega(x_1, x_2))$ where ω is the affine function depicted in the middle.

When two input data sets are linearly separable, this means that it becomes interesting to try to find a function ω which separates our two data sets. However, if we are to use a neural network to find it, we need something to strive for via optimization. Think of the hyperplane separating our two data sets (in Figure 1.8, this is the blue line) as the location where if we had a data point there, we would have no idea if it belongs to the category 0 or 1 ; in other words, the probability of belonging to either data set is set to 0.5. Well, this is exactly the value of the sigmoid function σ at 0, i.e. $\sigma(0) = \frac{1}{1+e^0} = 0.5$. Furthermore, σ maps any real number to a number between 0 and 1, so it would be nice if we could find parameters $w_1, \cdots, w_m, b$ such that the function $(x_1, \cdots, x_m) \mapsto \sigma(x_1 w_1 + \cdots + x_m w_m + b) = \sigma(x^\top w + b)$ was close to 0 (resp. 1) if the input $(x_1, \cdots, x_m)$ had output in the category 0 (resp. 1). But finding those parameters is exactly what a single-layer neural network is for (i.e. no hidden layer)! When the function $x \mapsto x^\top w + b$ equals zero, $\sigma(x^\top w + b) = 0.5$, so we don't know in which category the input data was belonging to. When $x^\top w + b$ is positive, the larger it is, the closer $\sigma(x^\top w + b)$ gets to 1 ; when it is negative, the larger it gets (in absolute value), the closer $\sigma(x^\top w + b)$ gets to 0. This seems like a good way to predict the output category from the input.

To sum things up, a logistic regression is a basic neural network with no hidden layers and bias in the input layer. The algorithm can also give a good model if the data is not necessarily linearly separable, but "almost" ; in other words, if there exists a line which puts almost all points in the category 0 on one side and almost all points of category 1 on the other side, logistic regression will approximate this line and be as accurate as the data allows it to be. See `linear-regression.py` for an example of this case using randomly generated data. Almost linearly separable data sets might occur in practice where such a model is a good approximation of reality but not a perfect one.

Non-linear logistic regression

To continue. See the code included in `GitHub` for an example.