**C. Description of the approach:**

1. **General information on neural networks:**

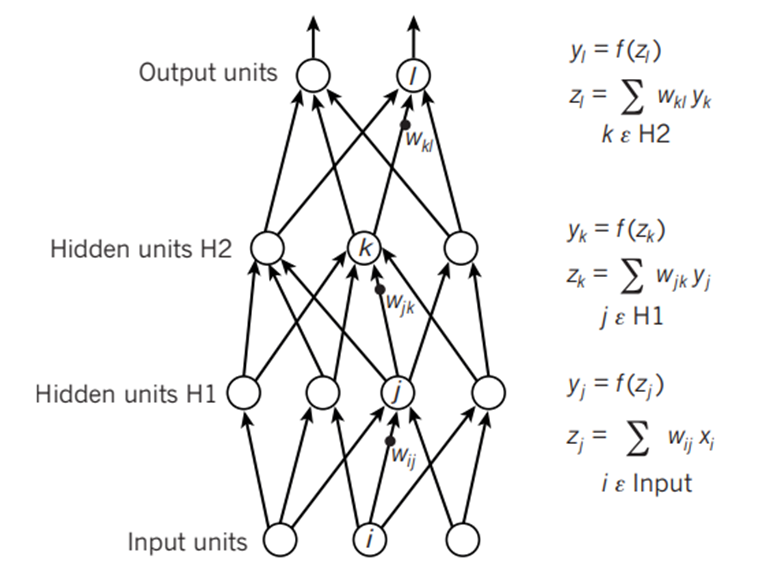
Neural networks were inspired by the biological neurons of the human brain. It is a set of interconnected formal neurons whose structure allows the resolution of complex problems.

This structure can be defined as a succession of layers, each of which takes its inputs from the output of the previous layer and where each layer is composed of a specific number of neurons, we speak of:

**Input layer** : which is the first layer that receives data as input.

**Output layer** : which is the last layer that returns the result.

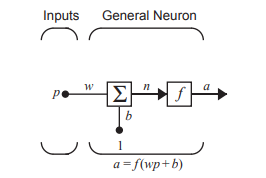
**Hidden layers :** which are all the layers between the input layer and the output layer.

**Weight ( weights ):** The links between neurons.

*Figure 1: Neural network*

Typically in the problems of classification one uses neural networks with one or 2 hidden layers, that proves to be sufficient.

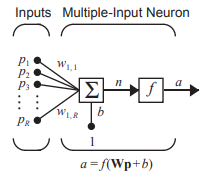
**Functional structure of a neuron:**



The input **p** is multiplied by the weight **w** and then the bias **b is added** . The result **n** passes through an activation function that produces the output of the neuron, **a** .

*Figure 2: Working structure of a*

*neuron with a single input*



Generally we have several inputs in a single neuron, so we will have:



Or more simply written:

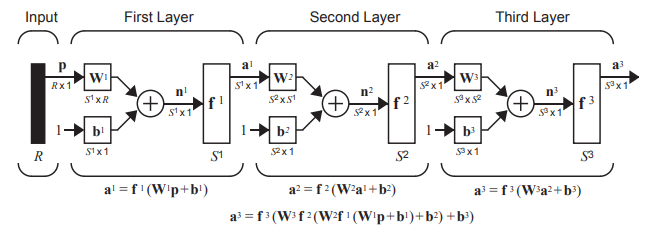


Which gives us:

*Figure 3: Functional structure of a*

*neuron with multiple inputs*

For a neural network with 4 hidden layers, we will therefore have a structure like this:



**Activation functions:**

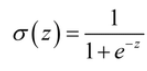
The activation function can be a linear or nonlinear function. A particular activation function is chosen to satisfy a specification of the problem the neuron is trying to solve.

Typically, a nonlinear differentiable activation function is used in the hidden layers of a neural network. This allows the model to learn more complex functions than a network trained using a linear activation function.

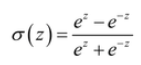
“In order to access a much richer space of assumptions that would benefit from deep representations, you need a nonlinearity or activation function.” —Page 72, Deep Learning with Python, 2017.

There are three activation functions that are widely used for hidden layers:

* Rectified Linear Activation ( **ReLU** ): f(z)=max(0,z)

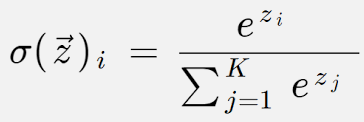


* Logistics ( **Sigmoid** ):



* Hyperbolic Tangent ( **Tanh** ):

The ReLU function remains the most widely used because it is both simple to implement and effective in overcoming the limitations of other previously popular activation functions, such as Sigmoid and Tanh .

Finally, we have the **softmax function** which is used at the level of the output layer in order to normalize the output vector, i.e. to have values between 0 and 1. It is defined as follows:

1. **Structure chosen:**

**Number of layers:** 4

Input layer: this layer will have 13 nodes, since we have 13 attributes for each person. Each input node will therefore receive an attribute.

Hidden layer 1: it will have 10 nodes. We will use the ReLU activation function .

Hidden layer 2: this hidden layer will have 2 nodes since we aim to have 2 classes thereafter. We will use the ReLU activation function .

Output layer: This layer will also have 2 nodes which would contain the final probability of belonging to one of the 2 classes. We will use a softmax activation function .

The values of the weights (w) and the biases (b) are chosen randomly.

1. **Coaching :**

The training consists of 3 main parts:

* Forward propagation: it consists of propagating the input forward through the network exactly as described in the functioning of a neuron.

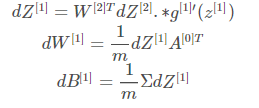
The input layer only contains the inputs so:

Then each subsequent layer will use the results of the previous layer to calculate its own results:

So on until you get to the final layer which will contain the results.

* Backward propagation: After completing a forward propagation of the data, we compare the results obtained from the last layer with the results we should have:

We calculate the error relating to: weight ( dW ), bias (dB) and activation function g ( dZ ) (here the numbers 0, 1 and 2 indicate the order of the layers, layer 0 comes before layer 1 )



A: matrix of layer 0 values

W: matrix of weights

g: enable function

* Parameter update: the parameters are updated as follows:

Parameter = parameter- alpha\*(estimated parameter error)

**D. Data pre-processing:**

The data is initially available on a csv file. The rows are relative to each person and the columns represent the attributes. The data has the dimensions (303 x 14)

We import the data from the csv file and transform it into a two-dimensional matrix (303 x 14) to be able to manipulate it.

We mix the data randomly before proceeding to divide them into 2 parts: 20% for the tests and 80% for the training. For each of the 2 matrices obtained, its transpose is calculated (to manipulate them more easily) then the vector containing the label of the expected result is isolated.

We obtain :

* Training matrix (13 x 243), label vector (1 x 243)
* Test matrix (13 x 60), label vector (1 x 60)

The rows of each matrix represent the attributes, so each column represents a person, which is suitable as an input to the neural network.

**For Code:**

We start by importing the data using the **pandas library** . We use the function **read\_csv ('file.csv').**

We perform the necessary operations (described above) to prepare the data using the **numpy library** . For the transformation of the data into a matrix we use the **array () function,** for the random shuffle the **random.shuffle () function** then for the calculation of the transpose of a matrix it is enough to add **.T** after the name of the matrix .

**E. Implementation:**

**1. Import the necessary libraries:**

We start by importing the libraries:

numpy : for vector manipulation

pandas: to import the data

matplotlib : for visualizations

import numpy as np

import pandas as pd

from matplotlib import pyplot as plt

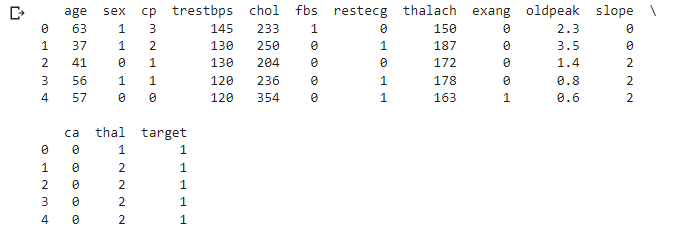
**2. Import and process data :**

We start by importing the data from the csv file and we try to visualize a few lines to see the structure of these.

data = pd.read\_csv ('dataset.csv')

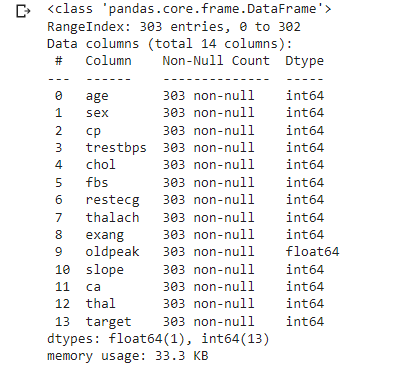
pd.set\_option (' display.max\_columns ', None)

print ( data.head ())



We display the details about each column, so we can check that there are no NULL values in which case we should replace them.

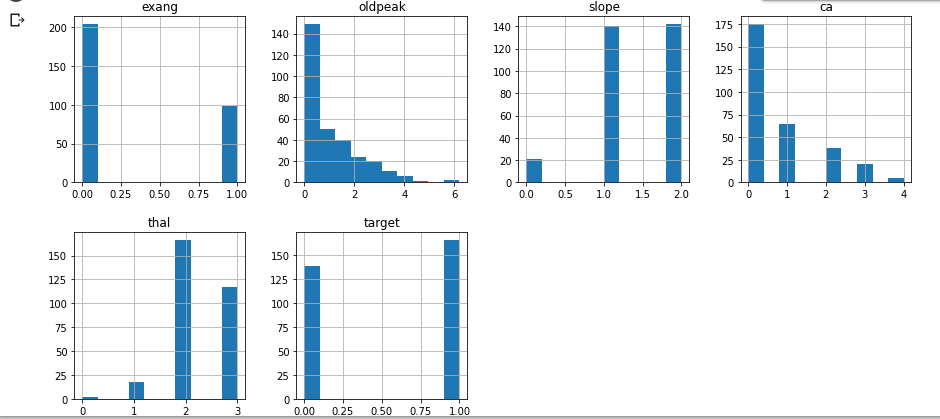
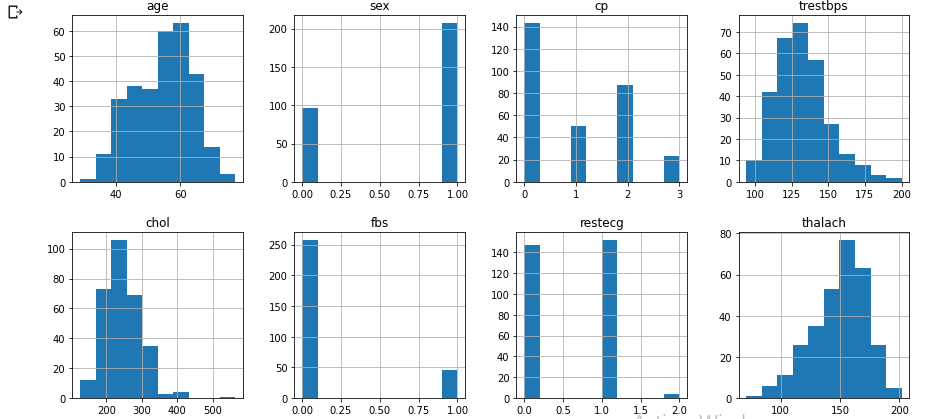
data.info( verbose = True )



We also take note of the magnitude and distribution of the data in relation to each attribute:

print ( data.describe ().T)

data.hist ( figsize =(20,20))



We begin the transformation of the recovered data into a matrix that we reorganize randomly.

data = np.array (data)

m, n = data.shape

print ("dimensions: (", m,"x ", n,")\n BEFORE:", data)

np.random.shuffle (data)

print ("\n AFTER:", data)

dimensions: (303 x 14)

BEFORE: [[63. 1. 3. ... 0. 1. 1.]

[37. 1. 2. ... 0. 2. 1.]

[41. 0.1. ... 0.2.1.]

...

[68. 1.0. ... 2.3.0.]

[57. 1.0. ... 1.3.0.]

[57. 0. 1. ... 1. 2. 0.]]

AFTER: [[58. 0. 1. ... 2. 2. 0.]

[46. 0.1. ... 0.2.1.]

[67. 0. 0. ... 2. 2. 1.]

...

[42. 1. 2. ... 0. 3. 1.]

[64. 1.0. ... 1.3.1.]

[52. 1.0. ... 0.0.0.]]

We then divide the generated matrix into 2 other matrices:

X\_dev : matrix which will contain the test data, the values of the objective class are on the other hand isolated towards the vector Y\_dev

X\_train : matrix that will contain the training data and similar, Y\_train will contain the values of the objective class

We calculate the transposes of these in such a way as to have the different attributes concerning each person on a column vector, which is suitable as input to the neural network.

data\_test = data[0:60].T

Y\_test = data\_test [n-1]

X\_test = data\_test [0:n-1]

data\_train = data[61:m].T

Y\_train = data\_train [n-1]

X\_train = data\_train [0:n-1]

**3. Training phase**

We define the activation functions: ReLU , softmax as well as the derivative of ReLU which will be used in back propagation

def read(Z):

return np.maximum (Z, 0)

def softmax (Z):

A = np.exp (Z) / sum ( np.exp (Z))

return A

def relu\_deriv (Z):

return Z > 0

We define our training function def train ( X , Y , alpha , iterations , random ) which will iterate between several executions of forward propagation and backward propagation and which will modify the parameters of the model each time to try to improve its performance. The parameter X designates the training matrix, Y the vector of targets , alpha the error modification parameter and finally random is a variable which will take the value 0 to perform a new training and the value 1 to load the parameters of a pre-trained model.

**Test on test data:**

def calculation\_result (X, W1, b1, W2, b2, W3, b3):

#Forward spread:

Z1 = W1.dot(X) + b1

A1 = reread(Z1)

Z2 = W2.dot(A1) + b2

A2 = reread(Z2)

Z3= W3.dot(A2) + b3

A3= softmax (Z3)

predictions = np.argmax (A3, 0)

return predictions

test\_res = calculation\_result ( X\_test , W1, b1, W2, b2, W3, b3)

#Exactness

print ( np.sum ( test\_res == Y\_test ) /Y\_test.size )

0.8666666666666667

**Verification examples:**

def test\_result (index, W1, b1, W2, b2, W3, b3):

cm = X\_train [:, index, None]

prediction = calculation\_result ( X\_train [:, index, None], W1, b1, W2, b2, W3, b3) c = int ( Y\_train [index])

print ("Prediction: ", prediction )

print ("Actual objective class: ", c)

test\_result (56, W1, b1, W2, b2, W3, b3 )

test\_result (70, W1, b1, W2, b2, W3, b3 )

test\_result (200, W1, b1, W2, b2, W3, b3 )

Prediction: [1]

Actual objective class: 1

Prediction: [0]

Actual objective class: 0

Prediction: [1]

Actual objective class: 1

**F. Analysis and results:**

We notice after many executions that due to the nature of the structure of the neural networks, more particularly the random initialization of the parameters, the performance of the model changes from one training to another. The results vary between 50% and 90% depending on the parameters used.

The weakest performances are very early observable at the beginning of the training when the first iterations generate accuracies as low as 40%, this indicates that the initialization is very unsuitable. These executions are therefore directly interrupted and a new training session is started. In general, the best performances are obtained for the first iterations which give accuracies of around 60%, these lead to accuracies as high as 87% at the end of the training.

With a parameter alpha=0.001 and a number of iterations of 10,000,000, we can have an accuracy of about 89%.

The test on the test data shows how well the model generalizes the predictions for different input data. We obtain accuracy rates of more than 85% so our model manages to generalize well the predictions from the training data, that is to say that even if we have new data, the model will give us mostly predictions correct.

**Influence of the Alpha parameter:**

The alpha parameter was initially initialized to 0.5 (value inspired by the state of the art), and then several values were tried: 0.8, 0.7, 0.6, 0.2, 0.1, 0.01, 0.001.

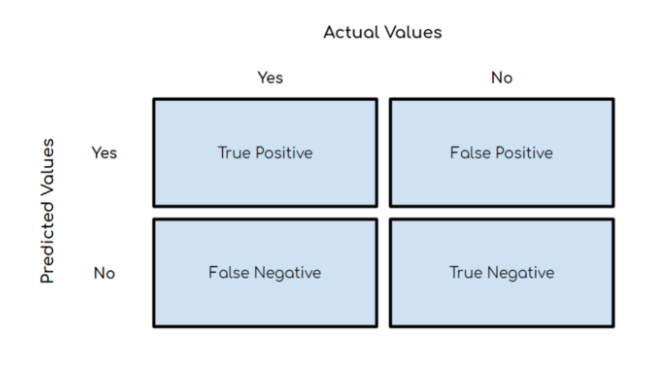
|  |  |
| --- | --- |
| Values | Remark |
| High values (0.8, 0.7, 0.6) | The finished model with generally poor performance. The model sometimes stalls on precise accuracy and no longer changes parameters regardless of the number of iterations remaining. |
| Small values (0.2, 0.1, 0.01, 0.001) | The smaller the value, the more the apprehension ability of the model increases. Small corrections are made from one iteration to another but the model blocks less in a fixed accuracy and learns better as long as the number of iterations is increased. |

**Influence of the number of iterations:**

|  |  |  |
| --- | --- | --- |
| Number of iterations | Result (average) | training time |
| 20 | Between 40% and 57% | <1s |
| 100 | Between 40% and 65% | <1s |
| 500 | Between 57% and 76% | 1s |
| 1000 | Between 57% and 84% | 2s |
| 100,000 | Between 61% and 84% | 4 mins |
| 2,000,000 | Between 80% and 86% | 18 mins |
| 6,000,000 | Between 87% and 90% | 45 mins |
| 10,000,000 | Between 88% and 90% | 1h5min |

**The confusion matrix:**

The confusion matrix is a way of judging the performance of the model in relation to our needs, for example if we are more interested in detecting sick people or if we prioritize the detection of non-sick people.



True positives: The number of sick people predicted correctly

False positive: The number of non-sick people predicted as sick

False negative: The number of sick people predicted as not sick

True Negatives : The number of people who are not sick correctly predicted

**Conclusion:**

Each problem can admit a precise neural network structure for its resolution and according to each problem the parameters are adjusted as the model performs.

We deduce for our case that with a fairly small alpha parameter, a satisfactory performance of the random parameters from the start of the training and a large enough number of iterations, this neural network gives good performance with respect to the classification of people who have heart problems or not because we end up with very well trained models that can reach 90% in accuracy.