



Lebanese University  
Faculty of Engineering III  
Department of Electrical and Electronic Engineering

# HEART DISEASE RISK PREDICTION

**A Machine Learning Project**

**Prepared by:** Hussein Awada (5505)

**Presented to:** Dr. Mohamed Awdi

Submitted for the Miniproject course

We would like to express our heartfelt gratitude to our supervisor Dr. Mohamed Awdi, for his constant support and him providing an excellent work in teaching us despite the difficult situations in our country and the world.

Most importantly, we thank Dr. Mohamed Awdi, to whom we have the honor to present this project, for the time and effort made to judge this work.

## **ABSTRACT**

The idea behind this project was to develop an AI based program in order to predict the risk of having a heart disease in the upcoming ten years to help in the early prognosis of these diseases.

The program consists of a neural network trained on a dataset of over 4,000 records and 15 attributes. Each attribute is a potential risk factor. There are both demographic, behavioral and medical risk factors.

Introduction.....	6
1. Concepts and Technologies Used.....	7
1.1. Python.....	7
1.2. Python libraries.....	7
1.3. Libraries to be imported in Python.....	7
1.3.1. NumPy.....	7
1.3.2. Pandas.....	7
1.3.3. Matplotlib .....	7
1.3.4. TensorFlow .....	8
1.4. Google Colaboratory .....	8
2. Deep Learning.....	9
2.1. Introduction.....	9
2.2. Defining Machine learning.....	9
2.3. Supervised Machine Learning.....	9
2.4. Neural Networks.....	9
2.4.1. Single Layer Perceptron (SLP) Model.....	10
2.4.2. Multilayer Perceptron Model (MLP).....	12
2.4.3. Output of Neurons.....	15
2.5. Activation Functions.....	16
2.6. Loss Function.....	18
2.7. Gradient Descent.....	19
2.8. Training with Gradient Descent.....	20
2.8.1. Gradient Descent for One Example.....	21
2.8.2. Gradient Descent for the Whole Training Set.....	22
3. The Application.....	24
3.1. Introduction.....	24
3.2. Setting up the Project.....	24
3.3. The Data.....	24
3.3.1. CSV Files .....	24
3.3.2. The Dataset.....	24
3.4. Preparing the Features and Labels.....	25
3.4.1. Inputting the Dataset .....	25
3.4.2. Filling the “Nan” values .....	25
3.4.3. Data Visualization .....	26
3.4.4. Dividing Data .....	26

3.4.5. Data Normalization .....	27
3.5. Building the Model .....	27
3.6. Training the Model.....	29
3.7. Training Results .....	29
3.8. Test Results .....	30
 4. Analysis and Conclusion.....	 31
4.1. Introduction.....	31
4.2. Discussions.....	31
4.2.1. The Training Accuracy .....	31
4.2.2. Overfitting .....	31
4.3. Conclusion.....	31
References.....	32

## **Introduction**

Cardiovascular diseases (CVDs), the leading cause of death globally, are a group of disorders of the heart and blood vessels and include coronary heart disease, cerebrovascular disease, rheumatic heart disease and other conditions. More than four out of five CVD deaths are due to heart attacks and strokes, and one third of these deaths occur prematurely in people under 70 years of age.

### **Key facts:**

- Cardiovascular diseases (CVDs) are the leading cause of death globally.
- An estimated 17.9 million people died from CVDs in 2019, representing 32% of all global deaths. Of these deaths, 85% were due to heart attack and stroke.
- Over three quarters of CVD deaths take place in low- and middle-income countries.
- Out of the 17 million premature deaths (under the age of 70) due to noncommunicable diseases in 2019, 38% were caused by CVDs.
- Most cardiovascular diseases can be prevented by addressing behavioral risk factors such as tobacco use, unhealthy diet and obesity, physical inactivity and harmful use of alcohol.
- It is important to detect cardiovascular disease as early as possible so that management with counselling and medicines can begin.

### **Solution?**

We decided to emphasize the contribution of the AI revolution towards such problem.

Our proposed solution is an application to predict whether a person is at a risk of having a heart disease based on previous collected records of a population. The application consists of neural network trained on the collected dataset.

# CHAPTER 1- CONCEPTS AND TECHNOLOGIES USED

## 1.1-Python:

Python is a widely used general-purpose, high-level programming language. It was initially designed by Guido van Rossum in 1991 and developed by Python Software Foundation. It was mainly developed for emphasis on code readability, and its syntax allows programmers to express concepts in fewer lines of code.

## 1.2-Python libraries:

A library is a collection of pre-combined codes that can be used iteratively to reduce the time required to code. There are over 137,000 python libraries present today. Python libraries play a vital role in developing machine learning, data science, data visualization, image and data manipulation applications and more.

## 1.3- Libraries to be imported in Python

### 1.3.1- NumPy

NumPy is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, Fourier transform, and matrices. NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely. NumPy stands for Numerical Python. NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently. This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

### 1.3.2- Pandas

Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

### 1.3.3- Matplotlib

John D. Hunter created Matplotlib, a plotting library for Python in 2003. It is one of the most widely used plotting libraries and among the first choices to plot graphs for quickly visualizing some data. Using Matplotlib, one can draw lots of graphs as per data like Bar Chart, Scatter Plot, Histograms, Contour Plots, Box Plot, Pie Chart, etc. one can also customize the labels, color, thickness of the plot details according to needs.

### **1.3.4- TensorFlow**

The primary software tool of deep learning is TensorFlow. It is an open source artificial intelligence library, using data flow graphs to build models. It allows developers to create large-scale neural networks with many layers. TensorFlow is mainly used for: Classification, Perception, Understanding, Discovering, Prediction and Creation.

“Keras” which is a deep learning API written in Python, runs on top of the machine learning platform TensorFlow.

### **1.4- Google Colaboratory**

Colaboratory, or “Colab” for short, is a product from Google Research. Colab allows anybody to write and execute arbitrary python code through the browser, and is especially well suited to machine learning, data analysis and education.



## CHAPTER 2- DEEP LEARNING

### 2.1- Introduction:

**Deep learning** is the subfield of machine learning that is devoted to building algorithms that explain and learn a high and low level of abstractions of data that traditional machine learning algorithms often cannot. The models in deep learning are often inspired by many sources of knowledge, such as game theory and neuroscience, and many of the models often mimic the basic structure of a human nervous system.

### 2.2- Defining Machine learning:

At the cross-section of statistics, mathematics, and computer science, machine learning refers to the science of creating and studying algorithms that improve their own behavior in an iterative manner by design. Machine learning is the practice of programming computers to learn from data.

### 2.3-Supervised Machine Learning:

In this type of machine-learning system the data fed into the algorithm, with the desired solution, that are referred to as “labels.” The goal is to approximate the mapping function so well that when you have new input data (X) that you can predict the output variables (Y) for that data. It is called supervised learning because the process of an algorithm learning from the training dataset can be thought of as a teacher supervising the learning process. We know the correct answers, the algorithm iteratively makes predictions on the training data and is corrected by the teacher. Learning stops when the algorithm achieves an acceptable level of performance.

### 2.4- Neural Networks:

Artificial neural networks are generally a chain of nodes associated with each other via the link from which they start interacting accordingly. Neurons perform operations and carry that result.

The optimum goal for a neural network is to find an optimized set of parameters (weights and biases) that best maps the features with their corresponding labels. The process of learning these parameters through a given dataset is called training.

Neural networks consist of:

- Input
- Output
- Weights and biases
- Activation function(s)

We cannot calculate the perfect weights for a neural network; there are too many unknowns. Instead, the problem of learning is cast as a search or optimization problem.

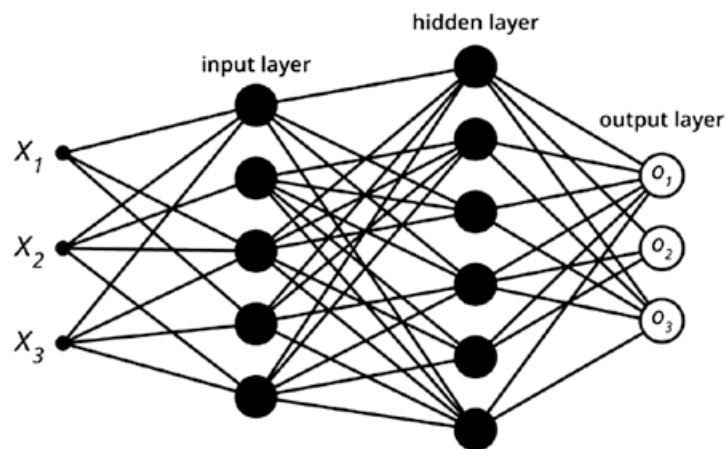


Figure 2.1-A Neural Network

### 2.4.1-Single Layer Perceptron (SLP) Model:

The simplest of the neural network models, SLP, was designed by researchers McCulloch and Pitts. In the eyes of many machine learning scientists, SLP is viewed as the beginning of artificial intelligence and provided inspiration in developing other neural network models and machine learning models. The SLP architecture is such that a single neuron is connected by many synapses, each of which contains a weight.

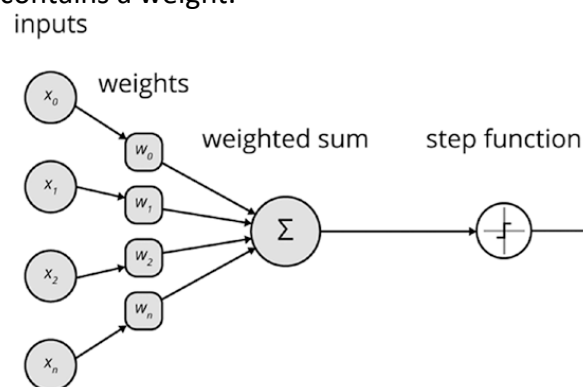


Figure 2.2-Single Neuron

The weights affect the output of the neuron. The aggregate values of the weights multiplied by the input are then summed within the neuron and then fed into an activation function, the standard function being the logistic function:

Let the vector of **inputs**  $[x_0, x_1, x_2, \dots, x_n]^T$

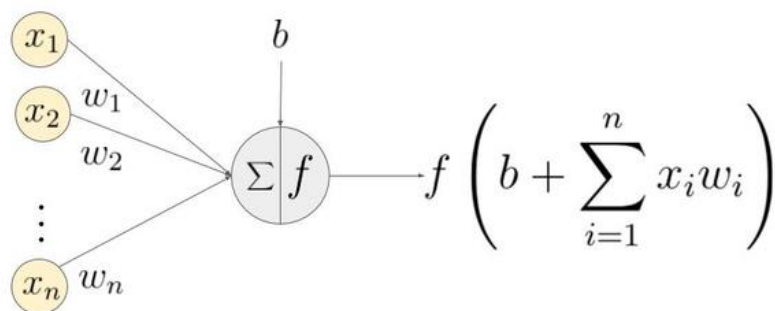
And the vector of **weights**  $[w_0, w_1, w_2, \dots, w_n]$

The output of the function is given by

$$y = f(x, w^T) = f\left(\sum_{i=0}^{i=n} x_i w_i\right)$$

Generally  $x_0 = 1$  and  $w_0 = b$ , where **b** is called the **bias**.

Then the scheme and equation of one neuron in a layer become:



**Figure 2.3-Single Neuron's Equation**

When dealing with big datasets, the number of features is large ( $n$  will be big), and so it is better to use a vector notation for the features and the weights, as follows:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{pmatrix}$$

Where we have indicated the vector with a boldfaced  $\mathbf{x}$ . For the weights, we use the same notation:

$$\mathbf{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix}$$

For consistency with formulas that we will use later, to multiply  $\mathbf{x}$  and  $\mathbf{w}$ , we will use matrix multiplication notation, and, therefore, we will write

$$\mathbf{w}^T \mathbf{x} = (w_1 \dots w_n) \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = w_1 x_1 + w_2 x_2 + \dots + w_n x_n$$

Where  $\mathbf{w}^T$  indicates the transpose of  $\mathbf{w}$ .  $\mathbf{z}$  can then be written with this vector notation as

$$\mathbf{z} = \mathbf{w}^T \mathbf{x}$$

And the neuron output  $\hat{\mathbf{y}}$  as

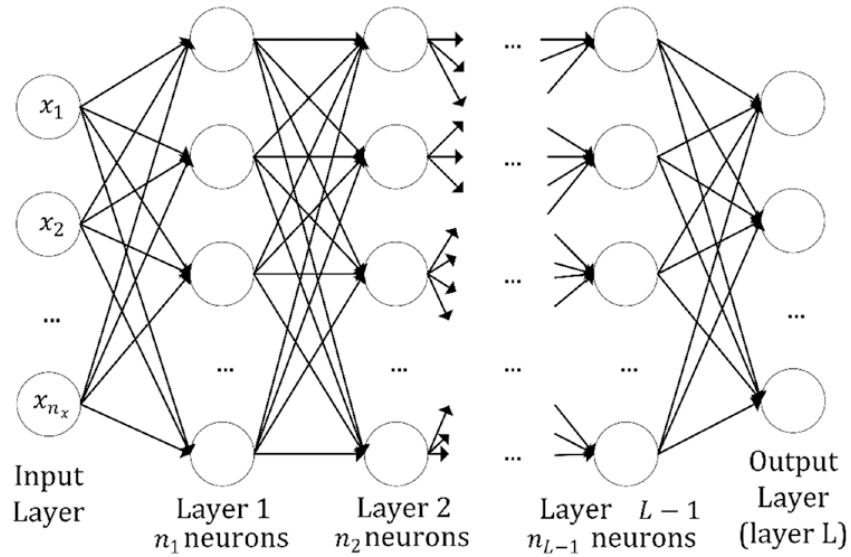
$$\hat{\mathbf{y}} = \mathbf{f}(\mathbf{z}) = \mathbf{f}(\mathbf{w}^T \mathbf{x} + \mathbf{b})$$

In brief,

- $\hat{\mathbf{y}} \rightarrow$  the neuron output
- $\mathbf{f}(\mathbf{z}) \rightarrow$  activation function (or transfer function) applied to  $\mathbf{z}$
- $\mathbf{w} \rightarrow$  weights
- $\mathbf{b} \rightarrow$  bias

### 2.4.2-Multilayer Perceptron Model (MLP):

Very similar to SLP, the *multilayer perceptron* (MLP) model features multiple layers that are interconnected in such a way that they form a feed-forward neural network. Each neuron in one layer has directed connections to the neurons of a separate layer.

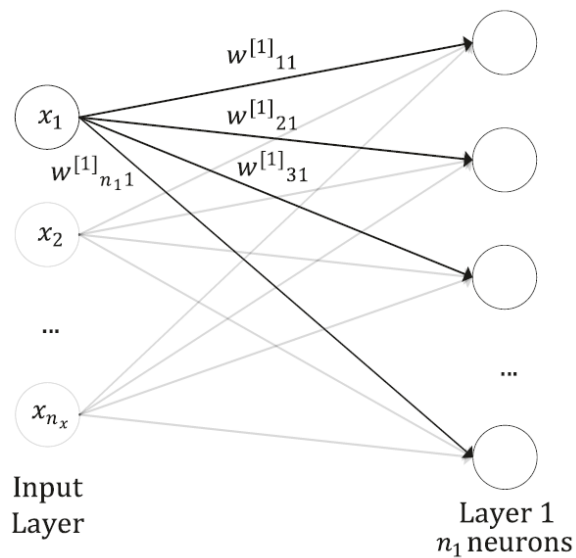


**Figure 2.4-Multi-Layer Neural Network**

To jump from one neuron to this is quite a big step. To build the model, we will have to work with matrix formalism, and therefore, we must get all the matrix dimensions right. First, here are some new notation.

- $L$ : Number of hidden layers, excluding the input layer but including the output layer
- $n_l$ : Number of neurons in layer  $l$

Where, by convention, we defined  $n_0 = n_x$ . Each connection between two neurons will have its own weight. Let's indicate the weight between neuron  $i$  in layer  $l$  and neuron  $j$  in layer  $l - 1$  with  $w_{ij}^{[l]}$ .



**Figure 1.5-First Hidden Layer**

The first two layers of a generic neural network, with the weights of the connections between the first neuron in the input layers and the others in the second layer. All other neurons and connections are drawn in light gray, to make the diagram clearer.

The weights and biases between the input layer and layer 1 can be written as a matrix, as follows:

$$\mathbf{W}^{[1]} = \begin{pmatrix} \mathbf{w}_{11}^{[1]} & \dots & \mathbf{w}_{1n_x}^{[1]} \\ \vdots & \ddots & \vdots \\ \mathbf{w}_{n_1 1}^{[1]} & \dots & \mathbf{w}_{n_1 n_x}^{[1]} \end{pmatrix}$$

$$\mathbf{b}^{[1]} = \begin{pmatrix} \mathbf{b}_1^{[1]} \\ \vdots \\ \mathbf{b}_{n_1}^{[1]} \end{pmatrix}$$

This means that our matrix  $\mathbf{W}^{[1]}$  has dimensions  $\mathbf{n}_1 \times \mathbf{n}_x$ . Of course, this can be generalized between any two layers  $\mathbf{l}$  and  $\mathbf{l} - 1$ , meaning that the weight matrix between two adjacent layers  $\mathbf{l}$  and  $\mathbf{l} - 1$ , indicated by  $\mathbf{W}^{[l]}$ , will have dimensions  $\mathbf{n}_l \times \mathbf{n}_{l-1}$ . By convention,  $\mathbf{n}_0 = \mathbf{n}_x$  is the number of input features (not the number of observations that we indicate with  $\mathbf{m}$ ).

$$\mathbf{W}^{[l]} = \begin{pmatrix} \mathbf{w}_{11}^{[l]} & \dots & \mathbf{w}_{1n_{l-1}}^{[l]} \\ \vdots & \ddots & \vdots \\ \mathbf{w}_{n_l 1}^{[l]} & \dots & \mathbf{w}_{n_l n_{l-1}}^{[l]} \end{pmatrix}$$

$$\mathbf{b}^{[l]} = \begin{pmatrix} \mathbf{b}_1^{[l]} \\ \vdots \\ \mathbf{b}_{n_l}^{[l]} \end{pmatrix}$$

**Note** the weight matrix between two adjacent layers  $\mathbf{l}$  and  $\mathbf{l} - 1$ , which we indicate with  $\mathbf{W}^{[l]}$ , will have dimensions  $\mathbf{n}_l \times \mathbf{n}_{l-1}$ , where, by convention,  $\mathbf{n}_0 = \mathbf{n}_x$  is the number of input features.

The bias will be a matrix this time. Remember that each neuron that receives inputs will have its own bias, so when considering our two layers,  $\mathbf{l}$  and  $\mathbf{l} - 1$ , we will require  $\mathbf{n}_l$  different values of  $\mathbf{b}$ . We will indicate this matrix with  $\mathbf{b}^{[l]}$ , and it will have dimensions  $\mathbf{n}_l \times \mathbf{1}$ .

**Note** The bias matrix for two adjacent layers  $\mathbf{l}$  and  $\mathbf{l} - 1$ , which we indicate with  $\mathbf{b}^{[l]}$ , will have dimensions  $\mathbf{n}_l \times \mathbf{1}$ .

### 2.4.3-Output of Neurons:

Now let's start considering the output of our neurons. To begin, we will consider the neuron of the first layer (remember that our input layer is by definition layer 0). Let's indicate its output with  $A^{[1]}$  and assume that all neurons in layer  $l$  use the same activation function, which we will indicate by  $g^{[1]}$ . Then we will have

$$A^{[1]} = g^{[1]}(Z^{[1]}) = g^{[1]}\left(\sum_{i=0}^{i=n_x} w_{ij}^{[1]}x_i + b_j^{[1]}\right)$$

As you can imagine, we want to have a matrix for all the output of layer 1, so we will use the notation

$$\begin{aligned} Z^{[1]} &= W^{[1]}X + b^{[1]} \\ Z^{[1]} &= \begin{pmatrix} z_1^{[1]} \\ \vdots \\ z_{n_1}^{[1]} \end{pmatrix} \\ A^{[1]} &= g^{[1]}(Z^{[1]}) = \begin{pmatrix} a_1^{[1]} \\ \vdots \\ a_{n_1}^{[1]} \end{pmatrix} \end{aligned}$$

Where  $Z^{[1]}$  will have dimensions  $n_1 \times 1$ , and where with  $X$ , we have indicated our matrix with all our observations (rows for the features, and columns for observations). We assume here that all neurons in layer  $l$  will use the same activation function that we will indicate with  $g^{[l]}$ .

We can easily generalize the previous equation for a layer  $l$

$$\begin{aligned} Z^{[l]} &= W^{[l]}A^{[l-1]} + b^{[l]} \\ Z^{[l]} &= \begin{pmatrix} z_1^{[l]} \\ \vdots \\ z_{n_l}^{[l]} \end{pmatrix} \end{aligned}$$

Because layer  $l$  will get its input from layer  $l - 1$ . We just need to substitute  $X$  with  $Z^{[l-1]}$ . Will have dimensions  $n_l \times 1$ . Our output in matrix form will then be

$$A^{[l]} = g^{[l]}(Z^{[l]})$$

$$\mathbf{A}^{[l]} = \begin{pmatrix} \mathbf{a}_1^{[l]} \\ \vdots \\ \mathbf{a}_{n_l}^{[l]} \end{pmatrix}$$

Where the activation function acts, as usual, element by element.

Following is a summary of the dimensions of all the matrices we have described so far:

- $\mathbf{W}^{[l]}$  has dimensions  $n_l \times n_{l-1}$ .
- $\mathbf{b}^{[l]}$  has dimensions  $n_l \times 1$ .
- $\mathbf{Z}^{[l-1]}$  has dimensions  $n_{l-1} \times 1$ .
- $\mathbf{Z}^{[l]}$ , has dimensions  $n_l \times 1$ .
- $\mathbf{A}^{[l]}$  has dimensions  $n_l \times 1$ .

$$\hat{\mathbf{Y}} = \mathbf{A}^{[L]}$$

## 2.5-Activation Functions:

An activation function is a function that is added into an artificial neural network in order to help the network learn complex patterns in the data. It takes in the output signal from the previous cell and converts it into some form that can be taken as input to the next cell.

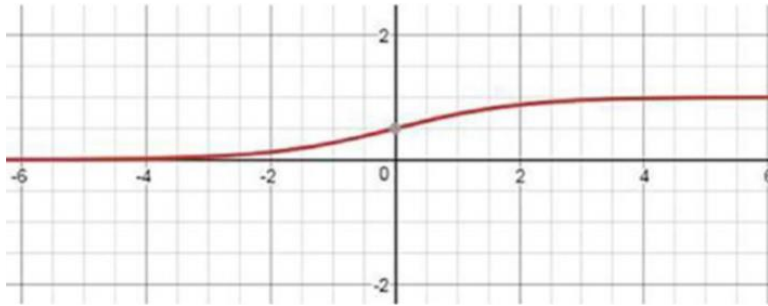
They also help in keeping the value of the output from the neuron restricted to a certain limit as per our requirement. This is important because input into the activation function is  $\mathbf{W}^* \mathbf{x} + \mathbf{b}$ . This value if not restricted to a certain limit can go very high in magnitude especially in case of very deep neural networks that have millions of parameters. This will lead to computational issues. For example, there are some activation functions (like sigmoid) that out specific values for different values of input (0 or 1).

The most important feature in an activation function is its ability to add non-linearity into a neural network.

There are many activation functions that are used for networks. Their usage depends on various factors, such as the interval where they are well-behaved (not saturated), how fast the function changes when its argument changes. Let's look at the one of the most frequently used activation functions; it's called *sigmoid*.

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





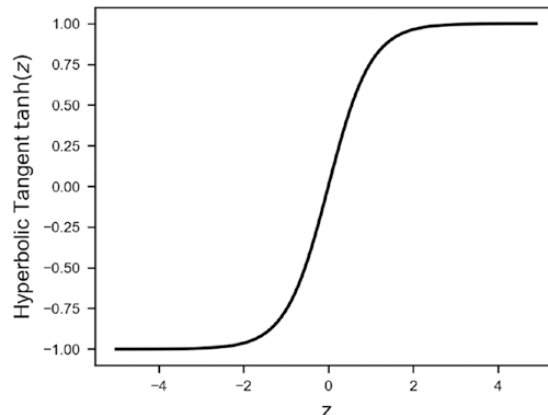
**Figure 2.6-The graph of the sigmoid function**

The sigmoid function (sometimes also called the *logistic* function) best behaves on the interval  $[-1, 1]$ . Outside of this interval, it quickly saturates, meaning that its value practically does not change with the change of its argument. That is why (as you will see in all this book's examples) the network's input data is typically normalized on the interval  $[-1, 1]$ .

Some activation functions are well-behaved on the interval  $[0, 1]$ , so the input data is correspondingly normalized on the interval  $[0, 1]$ .


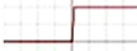


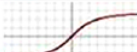




The hyperbolic tangent (Tanh) is also an s-shaped curve that goes from -1 to 1.

$$f(z) = \tanh(z)$$



**Figure 2.7-The graph of the hyperbolic tangent function**

**Figure 2.8** lists the most frequently used activation functions. It includes the function name, plot, equation, and derivative. This information will be useful when calculating various parts within a network.

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) <sup>(2)</sup>		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) <sup>(3)</sup>		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

*Figure 2.8-List of some activation functions and their derivatives*

## 2.6-Loss Function:

What Is a Loss Function and Loss?

In the context of an optimization algorithm, the function used to evaluate a candidate solution (i.e. a set of weights) is referred to as the objective function.

We may seek to maximize or minimize the objective function, meaning that we are searching for a candidate solution that has the highest or lowest score respectively.

Typically, with neural networks, we seek to minimize the error. As such, the objective function is often referred to as a cost function or a loss function and the value calculated by the loss function is referred to as simply “loss.”

The cost or loss function has an important job in that it must faithfully distill all aspects of the model down into a single number in such a way that improvements in that number are a sign of a better model. In calculating the error of the model during the optimization process, a loss function must be chosen.

This can be a challenging problem as the function must capture the properties of the problem and be motivated by concerns that are important to the project and stakeholders.

Our model here is a Binary Classification Problem, a problem where you classify an example as belonging to one of two classes.

The problem is framed as predicting the likelihood of an example belonging to class one, e.g. the class that you assign the integer value 1, whereas the other class is assigned the value 0.

**Output Layer Configuration:** One node with a sigmoid activation unit.

**Loss Function:** Cross-Entropy, also referred to as Logarithmic loss.

$$L(\hat{y}, y) = [(1 - y) \log(1 - \hat{y}) + y \log(\hat{y})]$$

Notice that the Cross entropy function is minimal when  $y = \hat{y}$

## 2.7-Gradient Descent:

Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. In machine learning, we use gradient descent to update the parameters of our model. Parameters refer to coefficients in Linear Regression and weights in neural networks. Gradient descent was originally proposed by Cauchy in 1847.

Gradient descent is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function.

$\mathbf{W} = \mathbf{W} - \alpha * \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ ; where  $\alpha$  is the learning Rate

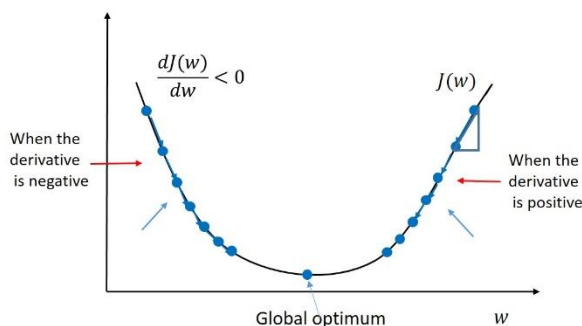


Figure 2.9-One dimension Gradient Descent

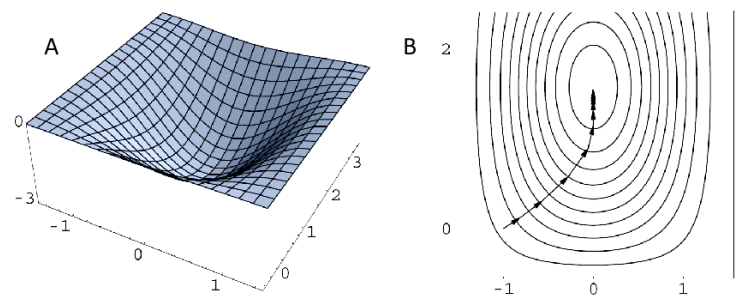


Figure 2.10-Two dimensions Gradient Descent

The size of each step is called the *learning rate*.

With a high learning rate we can cover more ground each step, but we risk overshooting the lowest point since the slope of the hill is constantly changing. With a very low learning rate, we can confidently move in the direction of the negative gradient since we are recalculating it so frequently. A low learning rate is more precise, but calculating the gradient is time-consuming, so it will take us a very long time to get to the bottom.

Taking small alphas will result in high number of iterations and high value can result in divergence.

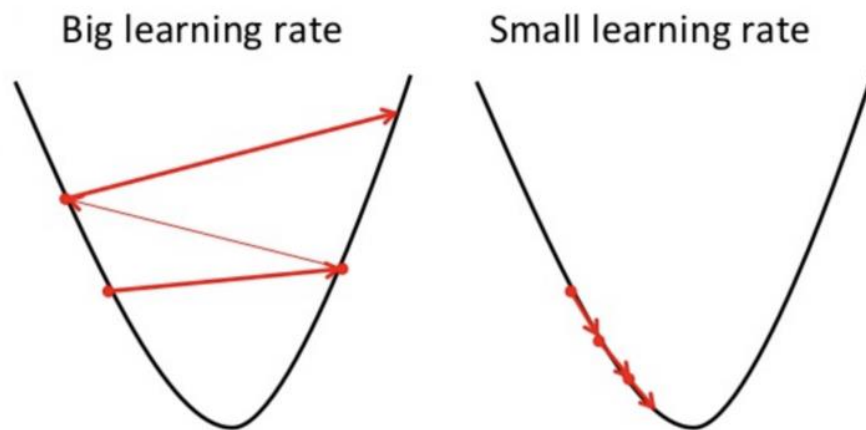


Figure 2.11-Small learning rate vs big learning rate

Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks.

## 2.8-Training with Gradient Descent:

Given a training set:

1. Initialize the Parameters  
For all layers  $l$  :
  - Set  $W^{[l]}$  at random
  - Set  $b^{[l]}$  to zeroes
2. Fix Learning Rate  $\alpha$
3. Set Gradients = 0  $\forall l$
4. Loop over the number of iterations:  
Compute  $\{A^{[2]}, \dots, A^{[L]}\}$  through forward propagation

Compute  $\delta^{[L]} = A^{[L]} - Y$   
 Compute errors  $\{\delta^{[L]}, \dots, \delta^{[2]}\}$   
 Compute gradients  $\frac{\partial L(\hat{y}, y)}{\partial W^{[l]}}, \frac{\partial L(\hat{y}, y)}{\partial b^{[l]}}$   
 For all layers  $l$  starting backward:
 

- $W^{[l]} = W^{[l]} - \alpha * \frac{\partial L(\hat{y}, y)}{\partial W^{[l]}}$
- $b^{[l]} = b^{[l]} - \alpha * \frac{\partial L(\hat{y}, y)}{\partial b^{[l]}}$

The following flowchart sums up the algorithm:

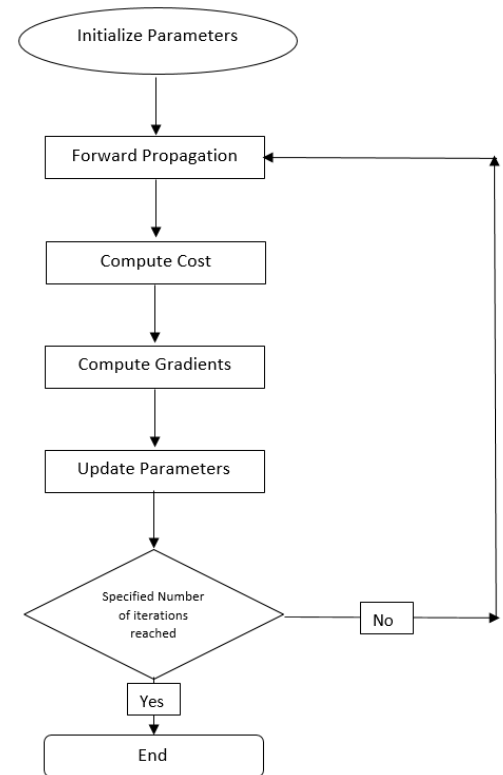


Figure 2.12-Gradient Descent Flowchart

### 2.8.1-Gradient Descent for One example:

We are working in a binary classification model so the size of the last layer is 1 and it takes values between 0 and 1.

Definitions:

- $z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$
- $a^{[l]} = g(z^{[l]})$
- $\delta^{[l]} = \frac{\partial L(\hat{y}, y)}{\partial z^{[l]}}$

- $L(\hat{y}, y) = [(1 - y) \log(1 - \hat{y}) + y \log(\hat{y})]$

The Gradients equations:

- $\delta^{[l]} = \frac{\partial L(\hat{y}, y)}{\partial z^{[l]}}$
- $\frac{\partial L(\hat{y}, y)}{\partial W^{[l]}} = \delta^{[l]} a^{[l-1]T}$
- $\frac{\partial L(\hat{y}, y)}{\partial b^{[l]}} = \frac{\partial L(\hat{y}, y)}{\partial z^{[l]}}$
- $\delta^{[l-1]} = \frac{\partial L(\hat{y}, y)}{\partial z^{[l-1]}} = W^{[l]T} \delta^{[l]} * g'(z^{[l-1]})$

Example: Last layer

- $\delta^{[l]} = \frac{\partial L(\hat{y}, y)}{\partial z^{[l]}} = \frac{\partial L(\hat{y}, y)}{\partial \hat{y}} * g'(z^{[l]})$ ; \*: Piecewise multiplication
- $\frac{\partial L(\hat{y}, y)}{\partial \hat{y}} = - \frac{\partial [(1-y) \log(1-\hat{y}) + y \log(\hat{y})]}{\partial \hat{y}} = \frac{1-y}{1-\hat{y}} - \frac{y}{\hat{y}} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}$
- $\delta^{[l]} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} * g'(z^{[l]}) = \frac{a^{[l]}-y}{g(z^{[l]})(1-g(z^{[l]}))} * (g(z^{[l]})(1-g(z^{[l]}))) = a^{[l]} - y$
- $\frac{\partial L(\hat{y}, y)}{\partial W^{[l]}} = \delta^{[l]} a^{[l-1]T} = (a^{[l]} - y) a^{[l-1]T}$
- $\frac{\partial L(\hat{y}, y)}{\partial b^{[l]}} = (a^{[l]} - y)$ 
  - $g(x) = \sigma(x) = \frac{1}{1+e^{-x}}$
  - $g'(x) = \sigma'(x) = \sigma(x)(1 - \sigma(x))$

## 2.8.2-Gradient Descent for the Whole Training Set:

Given:

- Training data:  $x^{(1)}, y^{(1)}, \dots, x^{(m)}, y^{(m)}$
- One training:  $x^{(i)} = (x_1^{(i)}, \dots, x_{n_x}^{(i)})$ , label :  $y^{(i)}$
- One forward pass through the network
  - Compute prediction  $\hat{y}^{(i)}$
- Loss function for one example
  - $L(\hat{y}, y) = [(1 - y) \log(1 - \hat{y}) + y \log(\hat{y})]$
- Loss function for training data
  - $J(W, b) = \frac{1}{N} \sum_i L(\hat{y}^{(i)}, y^{(i)})$

The whole training set is grouped in a matrix  $X$

$$X = \begin{pmatrix} x_1^{(1)} & \dots & x_1^{(m)} \\ \vdots & \ddots & \vdots \\ x_{n_x}^{(1)} & \dots & x_{n_x}^{(m)} \end{pmatrix}$$

Where the row dimension ( $m$ ) indicates the number of features of each sample and the column ( $n_x$ ) dimension indicates the number of samples.

The label vector become:  $Y = [y^{(1)}, \dots, y^{(m)}]$

And the predicted output vector becomes:  $\hat{Y} = [\hat{y}^{(1)}, \dots, \hat{y}^{(m)}]$

To compute the gradients of the weights and biases for a mini-batch, the gradients becomes:

- $L(\hat{Y}, Y) = [(1 - Y) \log(1 - Y) + Y \log(\hat{Y})]$
- $Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$
- $A^{[l]} = g(Z^{[l]})$
- **where:**

$$Z^{[l]} = \begin{pmatrix} z_1^{(1)} & \dots & z_1^{(m)} \\ \vdots & \ddots & \vdots \\ z_{n_l}^{(1)} & \dots & z_{n_l}^{(m)} \end{pmatrix}$$

$$A^{[l]} = \begin{pmatrix} a_1^{(1)} & \dots & a_1^{(m)} \\ \vdots & \ddots & \vdots \\ a_{n_l}^{(1)} & \dots & a_{n_l}^{(m)} \end{pmatrix}$$

- $Z^{[l-1]}$  has dimensions  $n_{l-1} \times m$ .
- $Z^{[l]}$ , has dimensions  $n_l \times m$ .
- $A^{[l]}$  has dimensions  $n_l \times m$ .

$$\frac{\partial L(\hat{Y}, Y)}{\partial W^{[l]}} = \frac{1}{m} * \sum_{i=1}^{i=m} \frac{\partial L(\hat{y}^{(i)}, y^{(i)})}{\partial W^{[l]}}$$

$$\frac{\partial L(\hat{Y}, Y)}{\partial b^{[l]}} = \frac{1}{m} * \sum_{i=1}^{i=m} \frac{\partial L(\hat{y}^{(i)}, y^{(i)})}{\partial b^{[l]}}$$

Definitions

- $\delta^{[l]} = \frac{\partial L(\hat{Y}, Y)}{\partial Z^{[l]}}$ ;  $\delta^{[l]}$  has the same dimention as  $Z^{[l]}$
- $\frac{\partial L(\hat{Y}, Y)}{\partial W^{[l]}} = (\delta^{[l]} A^{[l-1]T}) / m$
- $\frac{\partial L(\hat{Y}, Y)}{\partial b^{[l]}} = (\text{sum of rows of } \delta^{[l]}) / m$
- $\delta^{[l-1]} = \frac{\partial L(\hat{Y}, Y)}{\partial Z^{[l-1]}} = W^{[l]T} \delta^{[l]} * g'(Z^{[l-1]})$

## Chapter Three-The Application

### 3.1-Introduction:

In this chapter we will discuss the model with its components as well as the data it is built on.

### 3.2-Setting up the Project:

This project is written in Python using Google Colaboratory.

All python libraries needed are imported: NumPy, Pandas, Matplotlib, and TensorFlow.

### 3.3-The Data:

#### 3.3.1-CSV Files:

A Comma Separated Values (CSV) file is a plain text file that contains a list of data. These files are often used for exchanging data between different applications. For example, databases and contact managers often support CSV files.

These files may sometimes be called Character Separated Values or Comma Delimited files. They mostly use the comma character to separate (or delimit) data, but sometimes use other characters, like semicolons. The idea is that one can export complex data from one application to a CSV file, and then import the data in that CSV file into another application.

#### 3.3.2-The Dataset:

The dataset is publically available on the Kaggle website, and it is from an ongoing cardiovascular study on residents of the town of Framingham, Massachusetts. The classification goal is to predict whether the patient has 10-year risk of future coronary heart disease (CHD).The dataset provides the patients' information. It includes over 4,000 records and 15 attributes. Each attribute is a potential risk factor. There are both demographic, behavioral and medical risk factors.

Demographic:

- Sex: male or female (Nominal)
- Age: Age of the patient;(Continuous - Although the recorded ages have been truncated to whole numbers, the concept of age is continuous)

Behavioral

- Current Smoker: whether or not the patient is a current smoker (Nominal)



- Cigs Per Day: the number of cigarettes that the person smoked on average in one day.(can be considered continuous as one can have any number of cigarettes, even half a cigarette.)

Medical( history)

- BP Meds: whether or not the patient was on blood pressure medication (Nominal)
- Prevalent Stroke: whether or not the patient had previously had a stroke (Nominal)
- Prevalent Hyp: whether or not the patient was hypertensive (Nominal)
- Diabetes: whether or not the patient had diabetes (Nominal)

Medical(current)

- Tot Chol: total cholesterol level (Continuous)
- Sys BP: systolic blood pressure (Continuous)
- Dia BP: diastolic blood pressure (Continuous)
- BMI: Body Mass Index (Continuous)
- Heart Rate: heart rate (Continuous - In medical research, variables such as heart rate though in fact discrete, yet are considered continuous because of large number of possible values.)
- Glucose: glucose level (Continuous)

Predict variable (desired target)

- 10 year risk of coronary heart disease CHD (binary: “1”, means “Yes”, “0” means “No”)

## 3.4-Preparing the Features and Labels

### 3.4.1-Inputting the Dataset:

The dataset was uploaded to the google drive then loaded into the google colaboratory and stored in the variable “pd”.

```
df = pd.read_csv("/content/drive/MyDrive/Data/framingham.csv")
df.head()
```

	male	age	education	currentSmoker	cigsPerDay	BPMeds	prevalentStroke	prevalentHyp	diabetes	totChol	sysBP	diaBP	BMI	heartRate	glucose	TenYearCHD
0	1	39	4.0	0	0.0	0.0	0	0	0	195.0	106.0	70.0	26.97	80.0	77.0	0
1	0	46	2.0	0	0.0	0.0	0	0	0	250.0	121.0	81.0	28.73	95.0	76.0	0
2	1	48	1.0	1	20.0	0.0	0	0	0	245.0	127.5	80.0	25.34	75.0	70.0	0
3	0	61	3.0	1	30.0	0.0	0	1	0	225.0	150.0	95.0	28.58	65.0	103.0	1
4	0	46	3.0	1	23.0	0.0	0	0	0	285.0	130.0	84.0	23.10	85.0	85.0	0

### 3.4.2-Filling the “Nan” values:

We decided to fill the “Nan” values by the median of each attribute.

```
df['cigsPerDay'] = df['cigsPerDay'].fillna(df['cigsPerDay'].median())
df['BPMeds'] = df['BPMeds'].fillna(df['BPMeds'].median())
df['totChol'] = df['totChol'].fillna(df['totChol'].median())
df['BMI'] = df['BMI'].fillna(df['BMI'].median())
df['heartRate'] = df['heartRate'].fillna(df['heartRate'].median())
df['glucose'] = df['glucose'].fillna(df['glucose'].median())
df['education'] = df['education'].fillna(df['education'].median())
```

### 3.4.3-Data Visualization:

We chose to visualize the data by drawing the histograms of the attributes as well as the labels.

```
df.hist()
```

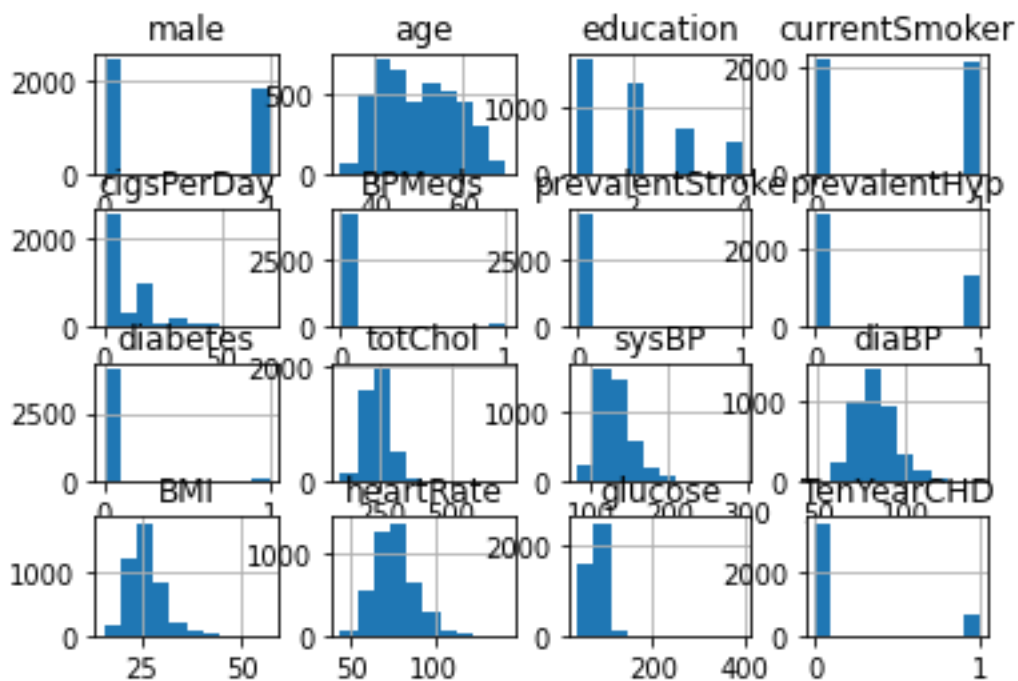


Figure 3.1-Histograms of the attributes of the dataset

### 3.4.4-Dividing Data:

The dataset is shuffle then divided into a training set (80% of the whole set) and a test set (20% of the whole set).

```
df = shuffle(df)
train_dataset = df.sample(frac=0.8, random_state=0)
test_dataset = df.drop(train_dataset.index)
```

Then for each set we separated the features (inputs) and the labels (outputs).

```
train_features = train_dataset.copy()
test_features = test_dataset.copy()

train_labels = train_features.pop('TenYearCHD')
test_labels = test_features.pop('TenYearCHD')
```

### 3.4.5- Data Normalization:

The preprocessing.Normalization layer is a clean and simple way to build that preprocessing into the model.

This layer will coerce its inputs into a distribution centered around 0 with standard deviation 1. It accomplishes this by precomputing the mean and variance of the data, and calling (input - mean)/sqrt(var) at runtime.

What happens in `adapt`: Compute mean and variance of the data and store them as the layer's weights.

```
normalizer = preprocessing.Normalization(axis=-1)
normalizer.adapt(np.array(train_features))
print(normalizer.mean.numpy())
```

```
[4.2861354e-01  4.9572556e+01  1.9769909e+00  4.9144554e-01  8.9289093e+00
 2.8318582e-02  6.1946898e-03  3.1209439e-01  2.6843660e-02  2.3677141e+02
 1.3226917e+02  8.2809746e+01  2.5797760e+01  7.5844238e+01  8.1856911e+01]
```

### 3.5-Building the Model:

The neural network chosen consists of:

- The input layer (size=15)
- 2 hidden layers with 64 neuron each with activation function hyperbolic tangent.
- The output layer with 1 neuron with activation function sigmoid so that the output between 0 and 1.

```

def build_and_compile_model(norm):
    model = keras.Sequential([
        norm,
        layers.Dense(64, activation='tanh'),
        layers.Dense(64, activation='tanh'),
        layers.Dense(1, activation='sigmoid')
    ])

    model.compile(loss='binary_crossentropy',
                  optimizer=tf.optimizers.Adam(learning_rate=0.00001), metrics=['accuracy'])
    return model

```

This function build the model using (keras.Sequential) with the specified requirements stated above and using the normalized input layer. Then it compile the model built using the “binary cross entropy” loss function and “Adam” as the optimizer with a learning rate=0.00001. The optimizer “Adam” here is responsible for the backpropagation algorithm. It is compiled also to return the accuracies will training.

Here we built the model and we can see a summary of its layers and parameters.

```

model = build_and_compile_model(normalizer)
model.summary()

```

Model: "sequential\_2"

Layer (type)	Output Shape	Param #
normalization_1 (Normalizati	(None, 15)	31
dense_6 (Dense)	(None, 64)	1024
dense_7 (Dense)	(None, 64)	4160
dense_8 (Dense)	(None, 1)	65
Total params: 5,280		
Trainable params: 5,249		
Non-trainable params: 31		

### 3.6-Training the Model:

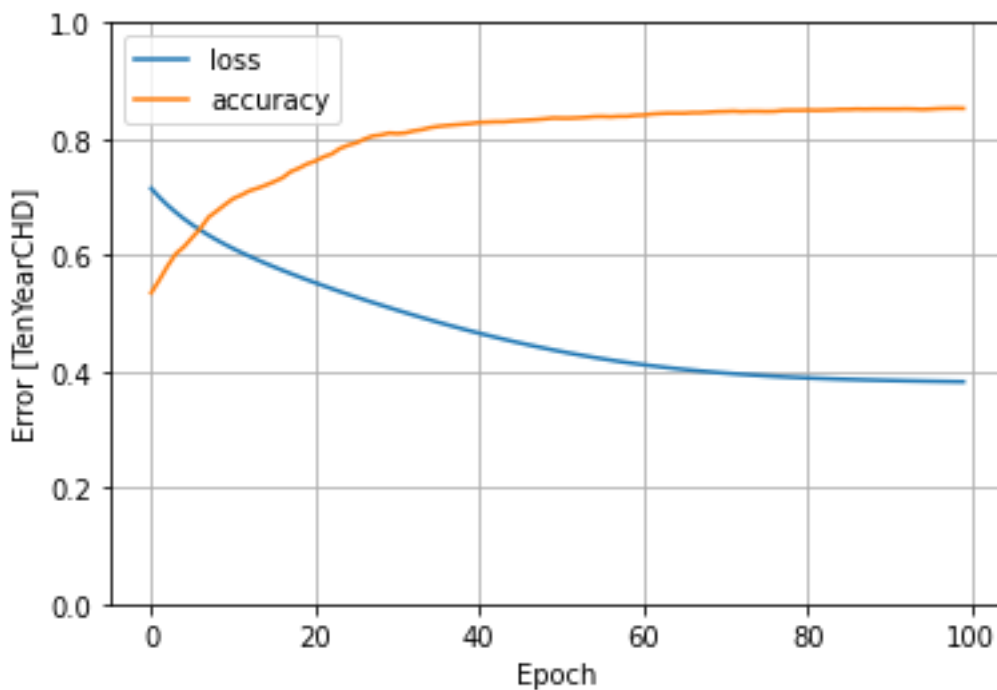
The training of the model is done using the “fit” function where it takes the training features, labels as well as the number of epochs. It will return a dictionary of the loss and accuracy on each epoch (here iteration).

```
history = model.fit(  
    train_features, train_labels, shuffle=True,  
    verbose=1, epochs=100)
```

An epoch means training the neural network with all the training data for one cycle. In an epoch, we use all of the data exactly once. A forward pass and a backward pass together are counted as one pass:

### 3.7-Training Results:

Over the whole number of batches, we can notice that the training loss is decreasing and the accuracy is increasing on the other hand.



*Figure 3.2-The variation of the training loss and accuracy as the number of epochs increase*

As we reach epoch 100, we can notice that the training loss reached 0.3822 and the training accuracy 0.8525.

```

Epoch 92/100
106/106 [=====] - 0s 2ms/step - loss: 0.3841 - accuracy: 0.8513
Epoch 93/100
106/106 [=====] - 0s 2ms/step - loss: 0.3838 - accuracy: 0.8516
Epoch 94/100
106/106 [=====] - 0s 2ms/step - loss: 0.3835 - accuracy: 0.8513
Epoch 95/100
106/106 [=====] - 0s 2ms/step - loss: 0.3833 - accuracy: 0.8507
Epoch 96/100
106/106 [=====] - 0s 1ms/step - loss: 0.3830 - accuracy: 0.8513
Epoch 97/100
106/106 [=====] - 0s 2ms/step - loss: 0.3828 - accuracy: 0.8522
Epoch 98/100
106/106 [=====] - 0s 2ms/step - loss: 0.3826 - accuracy: 0.8525
Epoch 99/100
106/106 [=====] - 0s 1ms/step - loss: 0.3824 - accuracy: 0.8528
Epoch 100/100
106/106 [=====] - 0s 2ms/step - loss: 0.3822 - accuracy: 0.8525

```

### 3.8-Test Results:

After training the model, we tested it on the rest set which we splitted before and we got a test accuracy of 0.8537 which very close to the training one.

```

▶ test_loss, test_acc = model.evaluate(test_features, test_labels, verbose=1)

print('Test accuracy:', test_acc)

27/27 [=====] - 0s 1ms/step - loss: 0.3804 - accuracy: 0.8538
Test accuracy: 0.8537735939025879

```

## **CHAPTER 4- Analysis and Conclusions**

### **4.1-Introduction:**

This chapter emphasizes on some notes about the application in addition to a conclusion.

### **4.2-Discussions:**

After using the model in the application, some points had to be noted such as:

#### **4.2.1-The Training Accuracy:**

Along the number of epochs we can notice the actual increase in accuracy for the training set till it reached 0.8525 which is an acceptable value.

#### **4.2.2- Overfitting:**

The test accuracy obtained 0.8537 which is very close to the accuracy obtained from training. This means that the model does the same behavior on data it hasn't seen before and thus the overfitting problem is solved here.

### **4.3-Conclusion:**

Predicting heart diseases is a very sophisticated problem and requires advanced technologies and equipments to perform perfectly. The simple neural network we trained did manage to do some predictions but not perfectly. More factors may interfere in a person having a risk of a heart disease or not, which make this program less reliable but still usable.

This application idea can definitely benefit from future improvements especially that it is related to a serious problem as heart diseases.

## **References:**

1. Michelucci, U. *Applied Deep Learning—A Case-Based Approach to Understanding Deep Neural Networks*; Apress Media, LLC: New York, NY, USA, 2018. ISBN 978-1-4842-3789-2.
2. Kůrková, Věra, et al., eds. *Artificial Neural Networks and Machine Learning—ICANN 2018: 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4-7, 2018, Proceedings, Part III*. Vol. 11141. Springer, 2018.
3. [Tensorflow.org](https://www.tensorflow.org)
4. [Keras API reference / Models API / Model training APIs](#)
5. [Keras API reference / Optimizers](#)
6. [All the Backpropagation derivatives](#)
7. [Deriving the Backpropagation Equations from Scratch by Thomas Kurbiel Towards Data Science](#)
8. [NumPy.org](https://numpy.org)
9. [Pandas.pydata.org](https://pandas.pydata.org)
10. [Framingham Dataset](#)