Sidi Mohamed Bin Abdullah University

National School of Applied Sciences in Fez

# Project Report

# AI

## **Topic :** Artificial Neural Networks

**E**mbedded **S**ystems and **I**ndustrial **D**ata **P**rocessing

**Submitted By :**

**CHERKANI HASSANI** Mohamed : n°7
**GHAFIFRI** Younes : n°16
**JADID** Taha : n°17
**OUBELLA** Mohamed : n°27

**Under The Guidance Of :**

Prof. **CHOUGRAD** Hiba

**April, 2020**

# ABSTRACT

The artificial neural network (ANN), or simply neural network, is a machine learning method evolved from the idea of simulating the human brain. The data explosion in Parkinson discovery research requires sophisticated analysis methods to uncover the hidden causes. The ANN is one of many versatile tools to meet the demand in diseases discovery modeling. Compared to a traditional regression approach, the ANN is capable of modeling complex nonlinear relationships. The ANN also has excellent fault tolerance and is fast and highly scalable with parallel processing. This project introduces the background of ANN development and outlines the basic concepts crucially important for understanding more sophisticated ANN. Several commonly used learning methods and network setups are discussed.

Furthermore, in order to overcome the limited and inadequate, sporadic symptom monitoring that Parkinson's disease (PD) patient suffer from, this project tends to be a useful tool allowing to healthcare professionals leading to precise medical decision making.

This approach will enable objective and remote monitoring of impaired motor function with the promise of profoundly changing the diagnostic, monitoring, and therapeutic landscape in PD.

# Table of Contents

# INTRODUCTION

Deep Learning is the most exciting and powerful branch of Machine Learning. It's a technique that teaches computers to do what comes naturally to humans: learn by example. Deep learning is a key technology behind driverless cars, enabling them to recognize a stop sign or to distinguish a pedestrian from a lamppost. It is the key to voice control in consumer devices like phones, tablets, TVs, and hands-free speakers. Deep learning is getting lots of attention lately and for good reason. It's achieving results that were not possible before.

In deep learning, a computer model learns to perform classification tasks directly from images, text, or sound. Deep learning models can achieve state-of-the-art accuracy, sometimes exceeding human-level performance. Models are trained by using a large set of labeled data and neural network architectures that contain many layers.

Deep Learning models can be used for a variety of complex tasks:

1. Artificial Neural Networks(ANN) for Regression and classification
2. Convolutional Neural Networks(CNN) for Computer Vision
3. Recurrent Neural Networks(RNN) for Time Series analysis
4. Self-organizing maps for Feature extraction
5. Deep Boltzmann machines for Recommendation systems
6. Auto Encoders for Recommendation systems

In this project, we will use **A**rtificial **N**eural **N**etworks or **ANN**. So first of all we must define **ANN**:

Artificial neural networks are one of the main tools used in machine learning. As the "neural" part of their name suggests, they are brain-inspired systems which are intended to replicate the way that we humans learn. Neural networks consist of input and output layers, as well as (in most cases) a hidden layer consisting of units that transform the input into something that the output layer can use. They are excellent tools for finding patterns which are far too complex or numerous for a human programmer to extract and teach the machine to recognize.

While neural networks (also called "perceptrons") have been around since the 1940s, it is only in the last several decades where they have become a major part of artificial intelligence. This is due to the arrival of a technique called "backpropagation," which allows networks to adjust their hidden layers of neurons in situations where the outcome doesn't match what the creator is hoping for — like a network designed to recognize dogs, which misidentifies a cat, for example.

# I- ARTIFICIAL NEURAL NETWORK

## 1. Artificial Neural Network presentation

Artificial Neural Networks or ANN is an information processing paradigm that is inspired by the way the biological nervous system such as brain process information. It is composed of large number of highly interconnected processing elements (neurons) working in unison to solve a specific problem.

### 1.1. Neurons

Biological Neurons (also called nerve cells) or simply neurons are the fundamental units of the brain and nervous system, the cells responsible for receiving sensory input from the external world via dendrites, process it and gives the output through Axons.

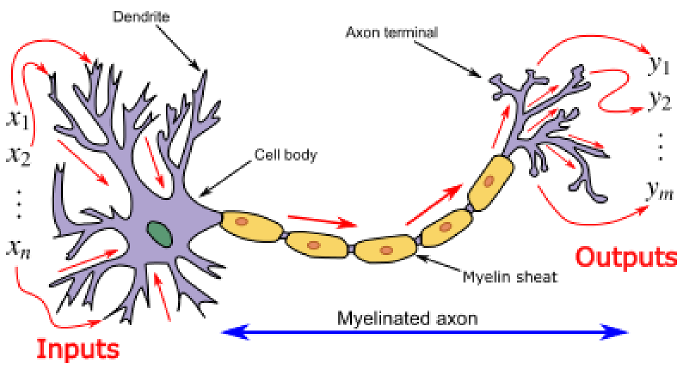


*Figure 1: A biological Neuron*

**Cell body (Soma):** The body of the neuron cell contains the nucleus and carries out biochemical transformation necessary to the life of neurons.

**Dendrites:** Each neuron has fine, hair-like tubular structures (extensions) around it. They branch out into a tree around the cell body. They accept incoming signals.

**Axon:** It is a long, thin, tubular structure that works like a transmission line.

**Synapse:** Neurons are connected to one another in a complex spatial arrangement. When axon reaches its final destination it branches again called terminal arborization. At the end of the axon are highly complex and specialized structures called synapses. The connection between two neurons takes place at these synapses.

Dendrites receive input through the synapses of other neurons. The soma processes these incoming signals over time and converts that processed value into an output, which is sent out to other neurons through the axon and the synapses.

The following diagram represents the general model of ANN which is inspired by a biological neuron. It is also called Perceptron.

A single layer neural network is called a Perceptron. It gives a single output.

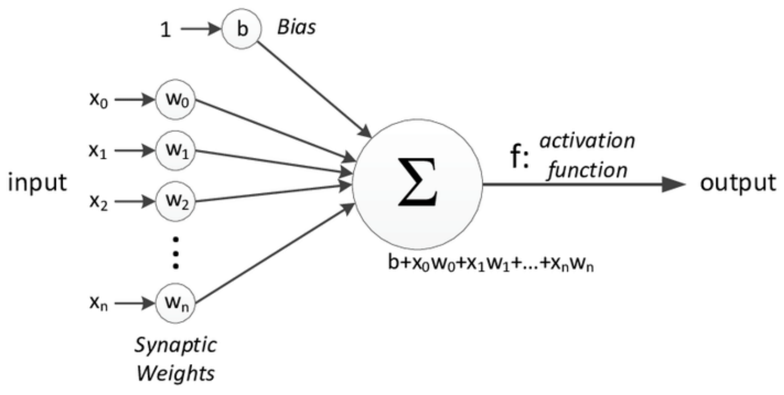


*Figure 2: Perceptron*

**Perceptron**

In the above figure, for one single observation, $x_0$, $x_1$, $x_2$, $x_3$...$x_{(n)}$ represents various inputs(independent variables) to the network. Each of these inputs is multiplied by a connection weight or synapse. The weights are represented as $w_0$, $w_1$, $w_2$, $w_3$….$w_{(n)}$ . **Weight shows the strength of a particular node.**

b is a bias value. A bias value allows you to shift the activation function up or down.

In the simplest case, these products are summed, fed to a transfer function (activation function) to generate a result, and this result is sent as output.

Mathematically, $x_1.w_1 + x_2.w_2 + x_3.w_3 ...... x_n.w_n = \sum x_i.w_i$

## 1.2. How does the Neural network work?

Let us take the example of the price of a property and to start with we have different factors assembled in a single row of data: Area, Bedrooms, Distance to city and Age.

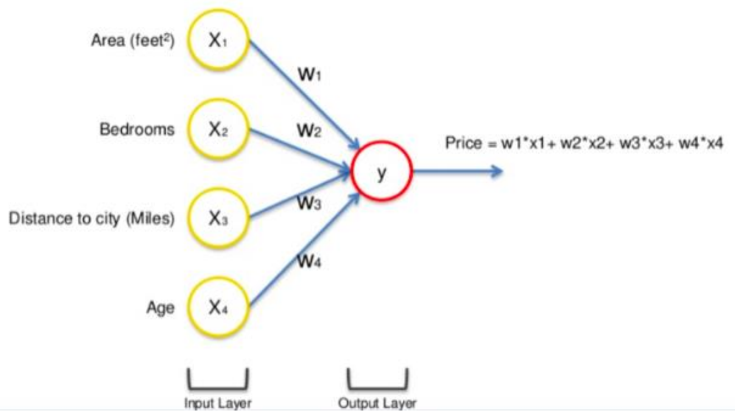


*Figure 3: Example of Neural Neutwork*

The input values go through the weighted synapses straight over to the output layer. All four will be analyzed, an activation function will be applied, and the results will be produced.

This is simple enough but there is a way to amplify the power of the Neural Network and increase its accuracy by the addition of a hidden layer that sits between the input and output layers.

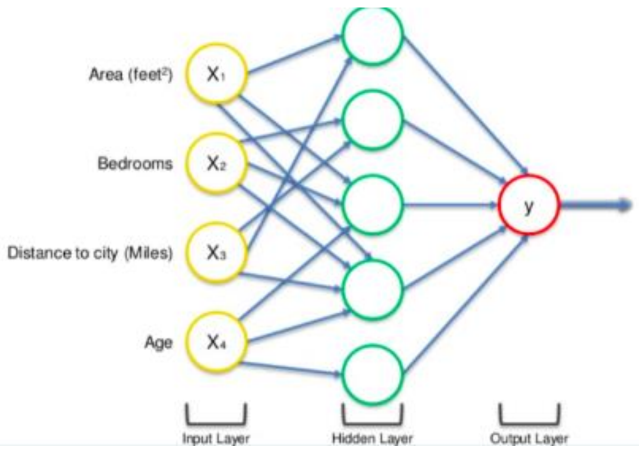


*Figure 4: Example with a Hidden Layer*

A neural network with a hidden layer(only showing non-0 values)

Now in the above figure, all 4 variables are connected to neurons via a synapse. However, not all of the synapses are weighted. they will either have a 0 value or non-0 value.

Here, the non-0 value → indicates the importance

0 value → They will be discarded.

Let's take the example of Area and Distance to City are non-zero for the first neuron, which means they are weighted and matter to the first neuron. The other two variables, Bedrooms and Age aren't weighted and so are not considered by the first neuron.

You may wonder why that first neuron is only considering two of the four variables. In this case, it is common on the property market that larger homes become cheaper the further they are from the city. That's a basic fact. So what this neuron may be doing is looking specifically for properties that are large but are not so far from the city.

Now, this is where the power of neural networks comes from. There are many of these neurons, each doing similar calculations with different combinations of these variables.

Once this criterion has been met, the neuron applies the activation function and do its calculations. The next neuron down may have weighted synapses of Distance to the city and, Bedrooms.

This way the neurons work and interact in a very flexible way allowing it to look for specific things and therefore make a comprehensive search for whatever it is trained for.

## 1.3. How do Neural networks learn?

Looking at an analogy may be useful in understanding the mechanisms of a neural network. Learning in a neural network is closely related to how we learn in our regular lives and activities — we perform an action and are either accepted or corrected by a trainer or coach to understand how to get better at a certain task. Similarly, neural networks require a trainer in order to describe what should have been produced as a response to the input. Based on the difference between the actual value and the predicted value, an error value also called **Cost Function** is computed and sent back through the system.

***Cost Function: One half of the squared difference between actual and output value.***

For each layer of the network, the cost function is analyzed and used to adjust the threshold and weights for the next input. Our aim is to minimize the cost function. The lower the cost function, the closer the actual value to the predicted value. In this way, the error keeps becoming marginally lesser in each run as the network learns how to analyze values.

We feed the resulting data back through the entire neural network. The weighted synapses connecting input variables to the neuron are the only thing we have control over.

As long as there exists a disparity between the actual value and the predicted value, we need to adjust those wights. Once we tweak them a little and run the neural network again, A new Cost function will be produced, hopefully, smaller than the last.

We need to repeat this process until we scrub the cost function down to as small as possible.

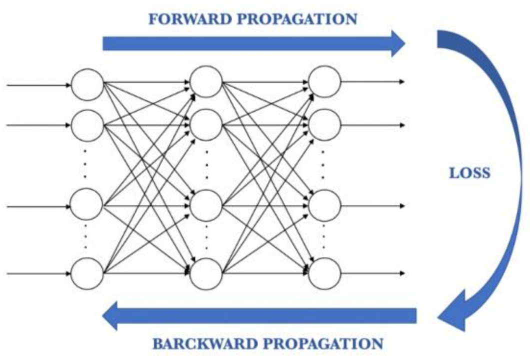


*Figure 5: Forward to Backward Propagation using LOSS*

The procedure described above is known as **Back-propagation** and is applied continuously through a network until the error value is kept at a minimum.

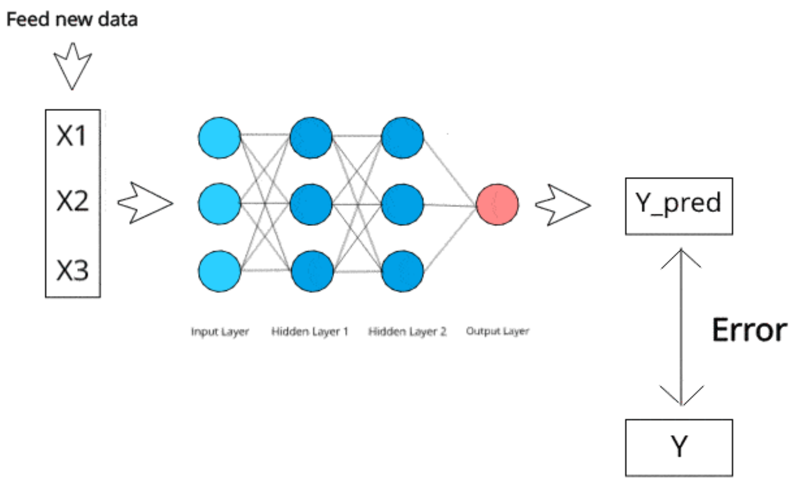


*Figure 6: Back-propagation*

## II. Activation functions

Neural network activation functions are a crucial component of deep learning. Activation functions determine the output of a deep learning model, its accuracy, and also the computational efficiency of training a model—which can make or break a large scale neural network. Activation functions also have a major effect on the neural network's ability to converge and the convergence speed, or in some cases, activation functions might prevent neural networks from converging in the first place.

The Activation function is important for an ANN to learn and make sense of something really complicated. Their main purpose is to convert an input signal of a node in an ANN to an output signal. This output signal is used as input to the next layer in the stack.

Activation function decides whether a neuron should be activated or not by calculating the weighted sum and further adding bias to it. The motive is to introduce non-linearity into the output of a neuron.

Activation functions are mathematical equations that determine the output of a neural network. The function is attached to each neuron in the network, and determines whether it should be activated ("fired") or not, based on whether each neuron's input is relevant for the model's prediction. Activation functions also help normalize the output of each neuron to a range between 1 and 0 or between -1 and 1.

If we do not apply activation function then the output signal would be simply linear function(one-degree polynomial). Now, a linear function is easy to solve but they are limited in their complexity, have less power. Without activation function, our model cannot learn and model complicated data such as images, videos, audio, speech, etc.

An additional aspect of activation functions is that they must be computationally efficient because they are calculated across thousands or even millions of neurons for each data sample. Modern neural networks use a technique called backpropagation to train the model, which places an increased computational strain on the activation function, and its derivative function.

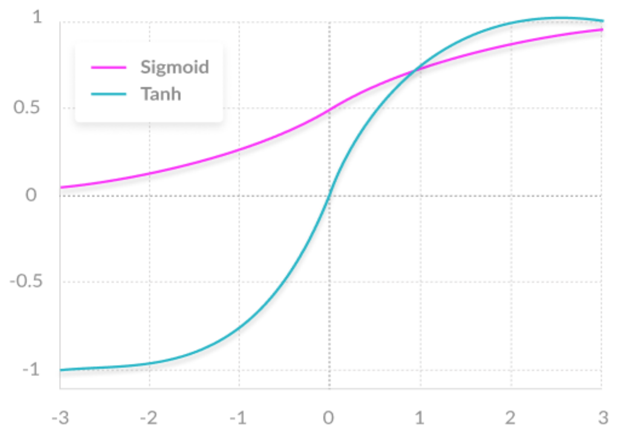


*Figure 7: Two Common neural network activation dunctions – Siemoid and Tanh*

### 1. Role of the Activation Function in a Neural Network Model

In a neural network, numeric data points, called inputs, are fed into the neurons in the input layer. Each neuron has a weight, and multiplying the input number with the weight gives the output of the neuron, which is transferred to the next layer.

The activation function is a mathematical “gate” in between the input feeding the current neuron and its output going to the next layer. It can be as simple as a step function that turns the neuron output on and off, depending on a rule or threshold. Or it can be a transformation that maps the input signals into output signals that are needed for the neural network to function.

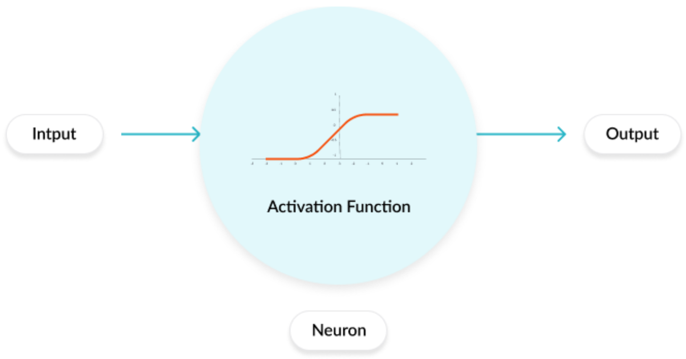


*Figure 8: Activation function*

Increasingly, neural networks use non-linear activation functions, which can help the network learn complex data, compute and learn almost any function representing a question, and provide accurate predictions.

## 2. Types of Activation Functions

### 2.1. Threshold Activation Function — (Binary step function)

A Binary step function is a threshold-based activation function. If the input value is above or below a certain threshold, the neuron is activated and sends exactly the same signal to the next layer.

$$f(x) = \begin{cases} 0 \text{ if } 0 > x \\ 1 \text{ if } x \geq 0 \end{cases}$$

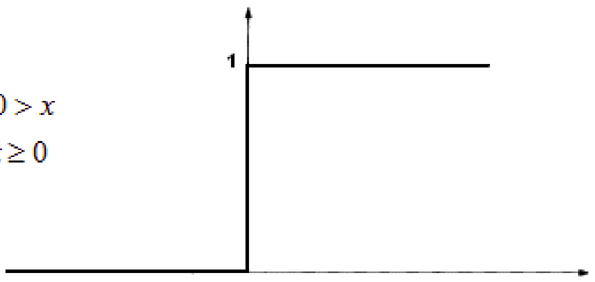


*Figure 9: Binary step function*

The problem with this function is for creating a binary classifier (1 or 0), but if you want multiple such neurons to be connected to bring in more classes, Class1, Class2, Class3, etc. In this case, all neurons will give 1, so we cannot decide.

### 2.2. Sigmoid Activation Function — (Logistic function)

A Sigmoid function is a mathematical function having a characteristic “S”-shaped curve or sigmoid curve which ranges between 0 and 1, therefore it is used for models where we need to predict the probability as an output.

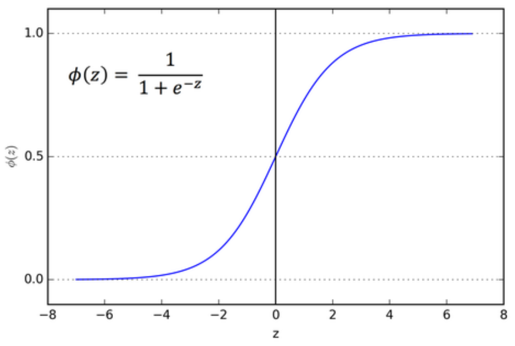


*Figure 10: Sigmoid function*

The Sigmoid function is differentiable, means we can find the slope of the curve at any 2 points.

The drawback of the Sigmoid activation function is that it can cause the neural network to get stuck at training time if strong negative input is provided.

## 2.3. Hyperbolic Tangent Function — (tanh)

It is similar to Sigmoid but better in performance. It is nonlinear in nature, so great we can stack layers. The function ranges between (-1,1).

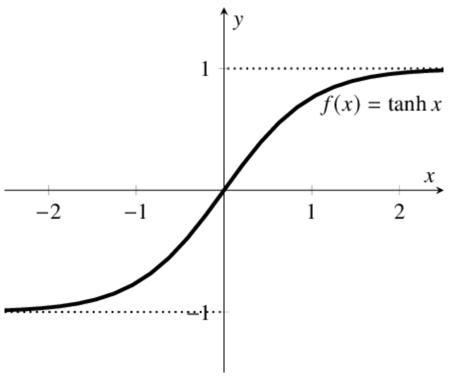


*Figure 11: tanh function*

The main advantage of this function is that strong negative inputs will be mapped to negative output and only zero-valued inputs are mapped to near-zero outputs.,So less likely to get stuck during training.

## 2.4. Rectified Linear Units — (ReLu)

ReLu is the most used activation function in CNN and ANN which ranges from zero to infinity.[0,∞[

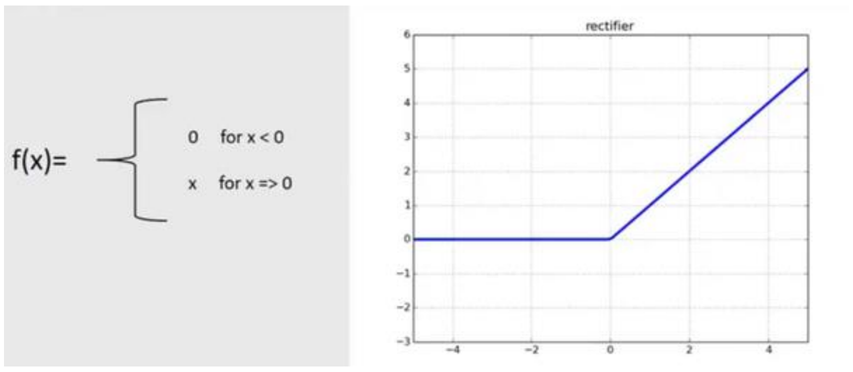


*Figure 12: ReLu*

It gives an output 'x' if x is positive and 0 otherwise. It looks like having the same problem of linear function as it is linear in the positive axis. Relu is non-linear in nature and a combination of ReLu is also non-linear. In fact, it is a good approximator and any function can be approximated with a combination of Relu.

ReLu is 6 times improved over hyperbolic tangent function.

It should only be applied to hidden layers of a neural network. So, for the output layer use softmax function for classification problem and for regression problem use a Linear function.

Here one problem is some gradients are fragile during training and can die. It causes a weight update which will make it never activate on any data point again. Basically ReLu could result in dead neurons.

To fix the problem of dying neurons, **Leaky ReLu** was introduced. So, Leaky ReLu introduces a small slope to keep the updates alive. Leaky ReLu ranges from -∞ to +∞.

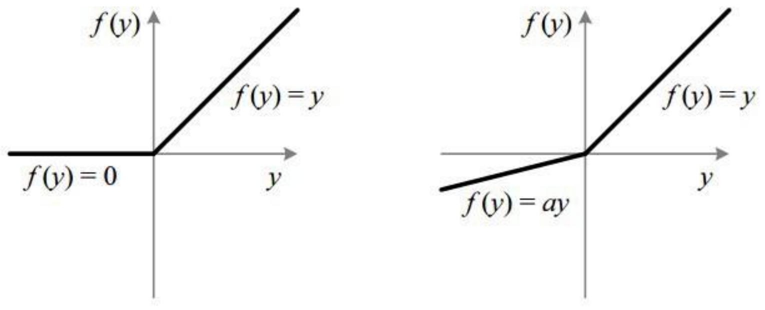


*Figure 13: Leaky ReLu*

Leak helps to increase the range of the ReLu function. Usually, the value of ***a*** = 0.01 or so.

When ***a*** is not 0.01, then it is called Randomized ReLu.

# III. Prevent overfitting using Dropout

Supervised machine learning is best understood as approximating a target function (**f**) that maps input variables (**X**) to an output variable (**Y**).

$$\mathbf{Y = f\,(X)}$$

This characterization describes the range of classification and prediction problems and the machine algorithms that can be used to address them.
An important consideration in learning the target function from the training data is how well the model generalizes to new data. Generalization is important because the data we collect is only a sample, it is incomplete and noisy

## 1. Overfitting

In statistics, **Overfitting** is "the production of an analysis that corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably".

In machine Learning, **Overfitting** happens when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data. This means that the noise or random fluctuations in the training data is picked up and learned as concepts by the model. The problem is that these concepts do not apply to new data and negatively impact the models ability to generalize.

When such an event occurs, the predictive model may give very good predictions on the data of the Training Set (the data it has already "seen" and to which it has adapted), but it will not predict so good on data that he had not yet seen during his learning phase.

So we can said predictive function does not generalize well. And that the model suffers from Overfitting.

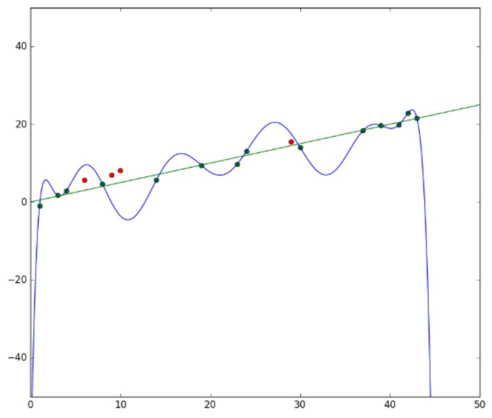


*Figure 14: Overfitting example*

The image above shows an example of **Overfitting**. The blue line represents a prediction function which passes through all the data in the Training Set (points in green). We can see that the function is unstable (large variance) and that it deviates a lot from the red dots which represent data not seen during the learning phase (Test Set).

## 2. How to limit Overfitting using Dropout

The most common problem in applied machine learning is Overfitting.

Overfitting is such a problem because the evaluation of machine learning algorithms on training data is different from the evaluation we actually care the most about, namely how well the algorithm performs on unseen data.

There are an important technique that we can use when evaluating machine learning algorithms to limit Overfitting is **Dropout**

### 2.1. Dropout

Dropout is a technique that addresses both these issues. It prevents Overfitting and provides a way of approximately combining exponentially many different neural network architectures efficiently. The term "dropout" refers to dropping out units (hidden and visible) in a neural network. By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections, as shown in *Figure 15*.

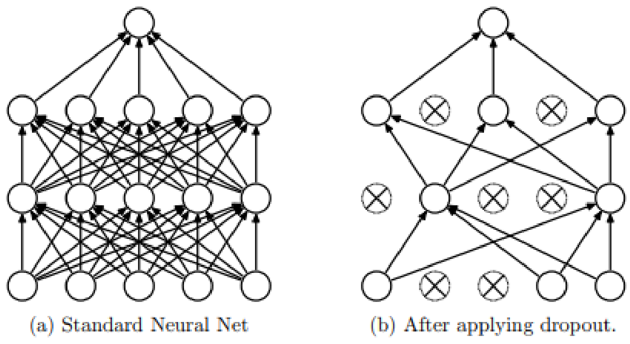


*Figure 15: Dropout Neural Net Model*

The choice of which units to drop is random. In the simplest case, each unit is retained with a fixed probability p independent of other units, where p can be chosen using a validation set or can simply be set at 0.5, which seems to be close to optimal for a wide range of networks and tasks. For the input units, however, the optimal probability of retention is usually closer to 1 than to 0.5.

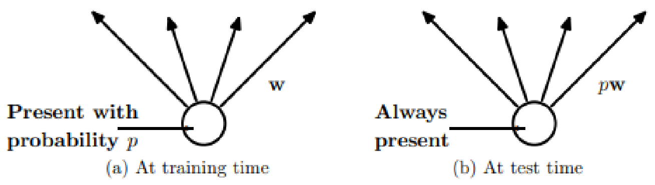


*Figure 16: Unit at training (a) and testing (b) time.*

Applying dropout to a neural network amounts to sampling a "thinned" network from it. The thinned network consists of all the units that survived dropout (*Figure 16.b*). A neural net with *n* units, can be seen as a collection of $2^n$ possible thinned neural networks. These networks all share weights so that the total number of parameters is still $O(n^2)$, or less. For each presentation of each training case, a new thinned network is sampled and trained. So training a neural network with dropout can be seen as training a collection of $2^n$ thinned networks with extensive weight sharing, where each thinned network gets trained very rarely, if at all.

At test time, it is not feasible to explicitly average the predictions from exponentially many thinned models. However, a very simple approximate averaging method works well in practice. The idea is to use a single neural net at test time without dropout. The weights of this network are scaled-down versions of the trained weights. If a unit is retained with probability p during training, the outgoing weights of that unit are multiplied by *p* at test time as shown in Figure 3. This ensures that for any hidden unit the expected output (under the distribution used to drop units at training time) is the same as the actual output at test time. By doing this scaling, $2^n$ networks with shared weights can be combined into a single neural network to be used at test time. We found that training a network with dropout and using this approximate averaging method at test time leads to significantly lower generalization error on a wide variety of classification problems compared to training with other regularization methods.

## a. Model Description

This section describes the dropout neural network model. Consider a neural network with L hidden layers. Let $l \in \{1, \dots, L\}$ index the hidden layers of the network. Let $\mathbf{z}^{(l)}$ denote the vector of inputs into layer l, $y^{(l)}$ denote the vector of outputs from layer l ($y^{(0)} = x$ is the input). $W^{(l)}$ and $b^{(l)}$ are the weights and biases at layer l. The feed-forward operation of a standard neural network (*Figure 17.a*) can be described as (for $l \in \{0, \dots, L-1\}$ and any hidden unit $i$)

$$\begin{aligned} z_i^{(l+1)} &= w_i^{(l+1)})\, y^l + b_i^{(l+1)}\, i\,, \\ y_i^{(l+1)} &= f(z_i^{(l+1)}), \end{aligned}$$

Where $f$ is any activation function, for example, $f(x) = 1/(1 + \exp(-x))$.
With dropout, the feed-forward operation becomes (*Figure 17.b*)

$$\begin{aligned} r_j^{(l)} &\sim Bernoulli(p), \\ \mathbf{Y}^{(l)} &= r^{(l)} * y^{(l)}, \\ z_i^{(l+1)} &= w_i^{(l+1)}\ \mathbf{Y}^l + b\ (l+1)\ i\,, \\ y_i^{(l+1)} &= f(z_i^{(l+1)}). \end{aligned}$$

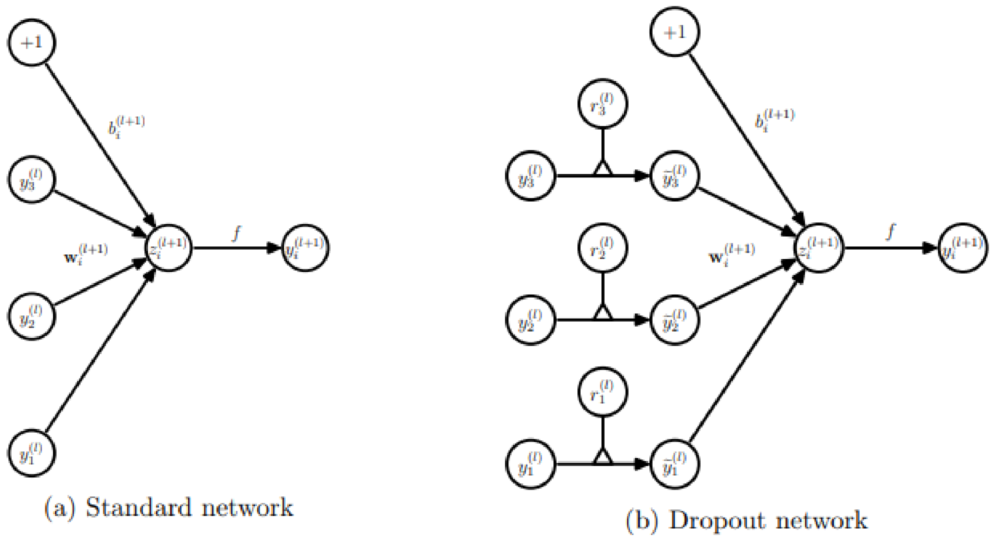


*Figure 17: Comparison of the basic operations of a standard and dropout network*

***NB:*** $\tilde{y}^{(l)}$ references to $\mathbf{Y}^{(l)}$ **in** *Figure 17*

For any layer l, $r^{(l)}$ is a vector of independent Bernoulli random variables each of which has probability *p* of being 1. This vector is sampled and multiplied element-wise with the outputs of that layer, $y^{(l)}$, to create the thinned outputs $\mathbf{Y}^{(l)}$ . The thinned outputs are then used as input to the next layer. This process is applied at each layer. This amounts to sampling a sub-network from a larger network. For learning, the derivatives of the loss function are backpropagated through the sub-network. At test time, the weights are scaled as $W_{\mathbf{test}}^{(l)} = pW^{(l)}$ as shown in (*Figure 16*). The resulting neural network is used without dropout.

*b.* *Learning Dropout Nets*

This section describes a procedure for training dropout neural nets.

**i. Backpropagation**

Dropout neural networks can be trained using stochastic gradient descent in a manner similar to standard neural nets. The only difference is that for each training case in a mini-batch, we sample a thinned network by dropping out units. Forward and backpropagation for that training case are done only on this thinned network. The gradients for each parameter are averaged over the training cases in each mini-batch. Any training case which does not use a parameter contributes a gradient of zero for that parameter. Many methods have been used to improve stochastic gradient descent such as momentum, annealed learning rates and L2 weight decay. Those were found to be useful for dropout neural networks as well

**ii. Unsupervised Pretraining**

Dropout can be applied to finetune nets that have been pretrained using these techniques. The pretraining procedure stays the same. The weights obtained from pretraining should be scaled up by a factor of 1/p. This makes sure that for each unit, the expected output from it under random dropout will be the same as the output during pretraining. We were initially concerned that the stochastic nature of dropout might wipe out the information in the pretrained weights. This did happen when the learning rates used during finetuning were comparable to the best learning rates for randomly initialized nets. However, when the learning rates were chosen to be smaller, the information in the pretrained weights seemed to be retained and we were able to get improvements in terms of the final generalization error compared to not using dropout when finetuning

## IV. Choice of loss functions: MSE, Binary_CrossEntropy, Ctegorical_CrossEntropy

Deep learning neural networks are trained using the stochastic gradient descent optimization algorithm. As part of the optimization algorithm, the error for the current state of the model must be estimated repeatedly. This requires the choice of an error function, conventionally called a **Loss function** that can be used to estimate the loss of the model so that the weights can be updated to reduce the loss on the next evaluation.

Neural network models learn a mapping from inputs to outputs from examples and the choice of **Loss function** must match the framing of the specific predictive modeling problem, such as classification or regression. Further, the configuration of the output layer must also be appropriate for the chosen loss function, where we can find the three Loss functions: Mean Square Error, Binary Cross Entropy, and Categorical Cross Entropy.

### 1. Mean Square Error (MSE)

The Mean Squared Error, or **MSE** loss is the default loss to use for regression problems.

Mathematically, it is the preferred loss function under the inference framework of maximum likelihood if the distribution of the target variable is Gaussian. It is the loss function to be evaluated first and only changed if you have a good reason.

$$MSE = \frac{1}{n} \Sigma \underbrace{\left( y - \widehat{y} \right)^2}_{\text{The square of the difference between actual and predicted}}$$

*Figure 18: Equation of MSE*

Mean squared error is calculated as the average of the squared differences between the predicted and actual values. The result is always positive regardless of the sign of the predicted and actual values and a perfect value is 0.0. The squaring means that larger mistakes result in more error than smaller mistakes, meaning that the model is punished for making larger mistakes.

#### 1.1. Example

For Example, we have a neural network which takes house data and predicts house price. In this case, you can use the MSE loss. Basically, in the case

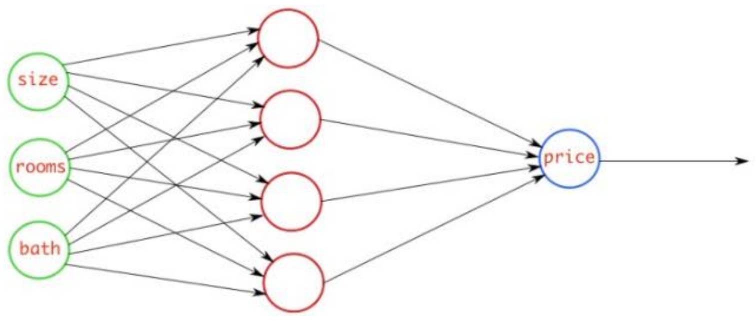


*Figure 19: Example MSE*

#### 1.2. Application

The mean squared error loss function can be used in Keras by specifying ‘*mse*‘ or ‘*mean_squared_error*‘ as the loss function when compiling the model.

**model.compile(** *loss='mean_squared_error'* **)**

In this case, we can see that the model resulted in slightly worse MSE on both the training and test dataset. It may not be a good fit for this problem as the distribution of the target variable is a standard Gaussian.

*Train: 0.165, Test: 0.184*

A line plot is also created showing the mean squared logarithmic error loss over the training epochs for both the train (blue) and test (orange) sets (top), and a similar plot for the mean squared error (bottom). We can see that the MSLE converged well over the 100 epoch's algorithm; it appears that the MSE may be showing signs of **Overfitting** the problem, dropping fast and starting to rise from epoch 20 onward

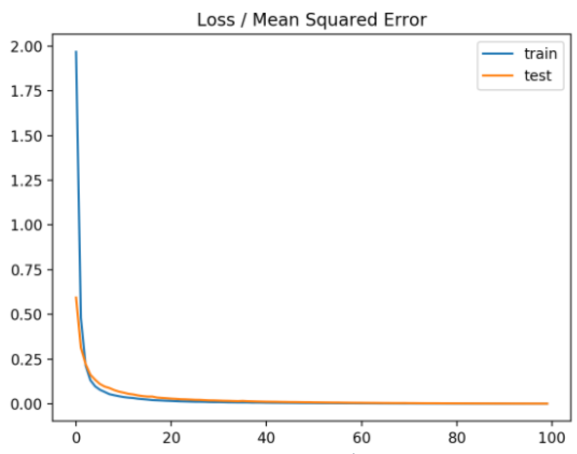


*Figure 20: Plot MSE*

## 2. Binary Cross Entropy (BCE)

Binary Cross-entropy is the default loss function to use for binary classification problems.
We can use it just when we have a binary classification task, one of the loss function you can go ahead is this one. If you are using *BCE* loss function, you just need one output node to classify the data into two classes. The output value should be passed through a *sigmoid* activation function and the range of output is (0 – 1).
In a binary classification algorithm such as Logistic regression, the goal is to minimize the cross-entropy function.

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

Binary Cross-Entropy / Log Loss

*Figure 21: Equation of BCE*

### 2.1. Example

For example, we have a neural network which takes atmosphere data and predicts whether it will rain or not. If the output is greater than 0.5, the network classifies it as rain and if the output is less than 0.5, the network classifies it as not rain. (it could be opposite depending upon how you train the network). More the probability score value, more the chance of raining.

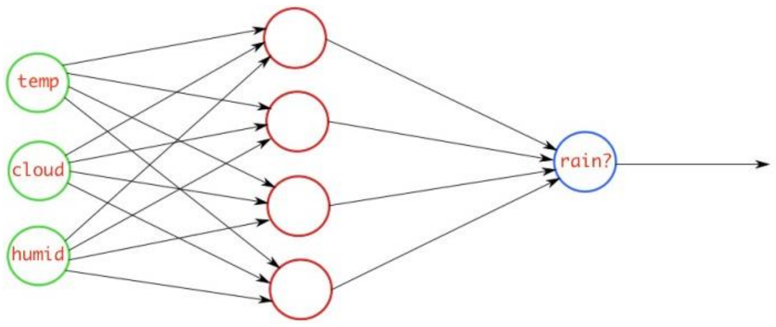


*Figure 22: Example BCE*

While training the network, the target value fed to the network should be 1 if it is *raining* otherwise 0.

## 2.2. Application

Cross-entropy can be specified as the loss function in Keras by specifying '*binary_crossentropy*' when compiling the model.

```
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
```

In this case, we can see that the model learned the problem reasonably well, achieving about **83%** accuracy on the training dataset and about **85%** on the test dataset.

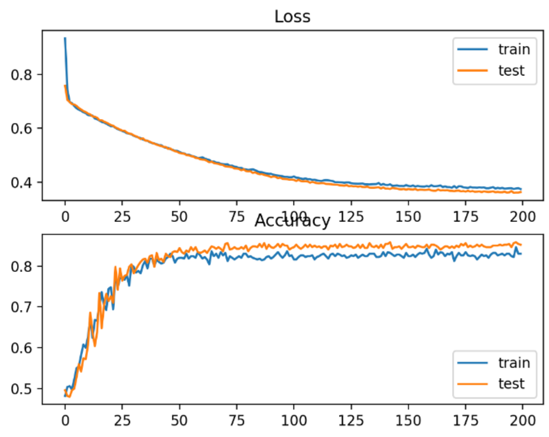


*Figure 23: Plot BCE*

## 3. Categorical Cross Entropy (CCE)

We can use the Categorical Cross Entropy when we have a multi-class classification task. If you are using CCE loss function, there must be the same number of output nodes as the classes. And the final layer output should be passed through a *softmax* activation so that each node output a probability value between (0–1).

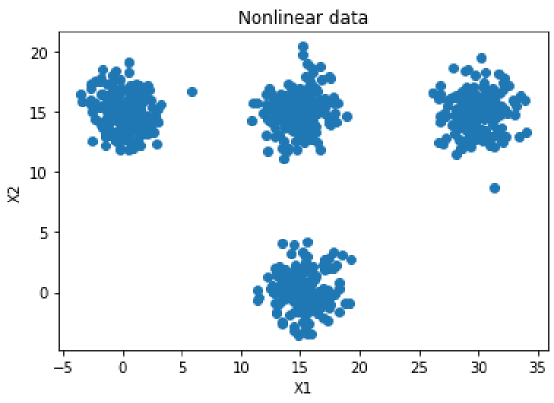


*Figure 24: Classification of Data Set CCE*

As you can see in the *Figure 24*, we have classified our Data Set to 4 type of shapes, that's means 4 categories of Data

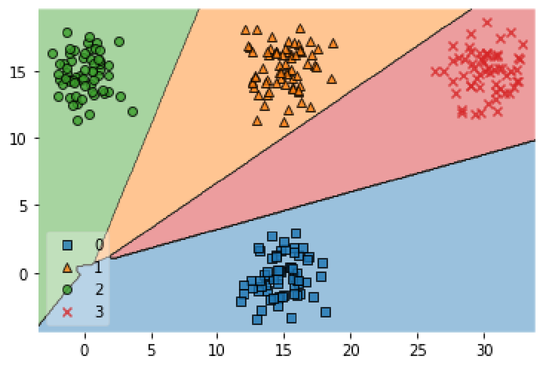


*Figure 25: Model CCE*

And we generate a modal as showed in Figure 10, with 4 classes

## 3.1. Example

For example, we have a neural network which takes an image and classifies it into a cat or dog. If cat node has high probability score then the image is classified into cat otherwise dog. Basically, whichever class node has the highest probability score, the image is classified into that class.

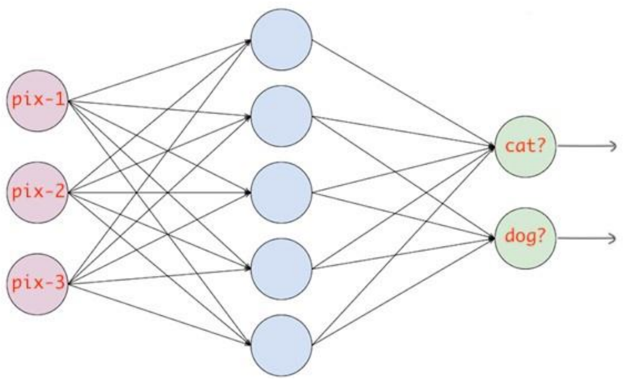


Figure 26: Example CCE

For feeding the target value at the time of training, we have to one-hot encode them. If the image is of cat then target vector would be (1, 0) and if the image is of dog, target vector would be (0, 1). Basically, target vector would be of the same size as the number of classes and the index position corresponding to the actual class would be 1 and all others would be zero.

## 3.2. Application

Cross-entropy can be specified as the loss function in Keras by specifying '*binary_crossentropy*' when compiling the model.

**model.compile**(*loss= 'categorical_crossentropy',optimizer=keras.optimizers.adam(lr=0.001),metrics=['accuracy']*)

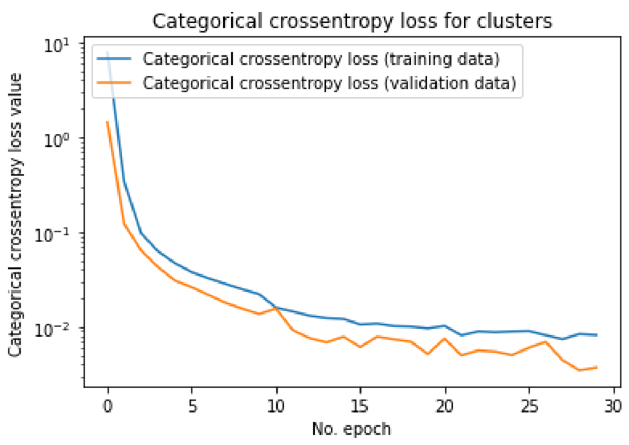


Figure 27: Plot CCE

## V. Stochastic gradient descent:

As we explained before, an ANN learning, essentially means finding the best values for all weights and biases in the network, to best estimate a target function (f) that maps input data (X) onto output variables (Y). of course, this describes all classification and regression problems.

Calculating the minima of the loss function (already explained at "ChapitreI : 1.3. How Do Neural Networks learn" ) allows us to find the most optimized value for these variables (weights and biases).

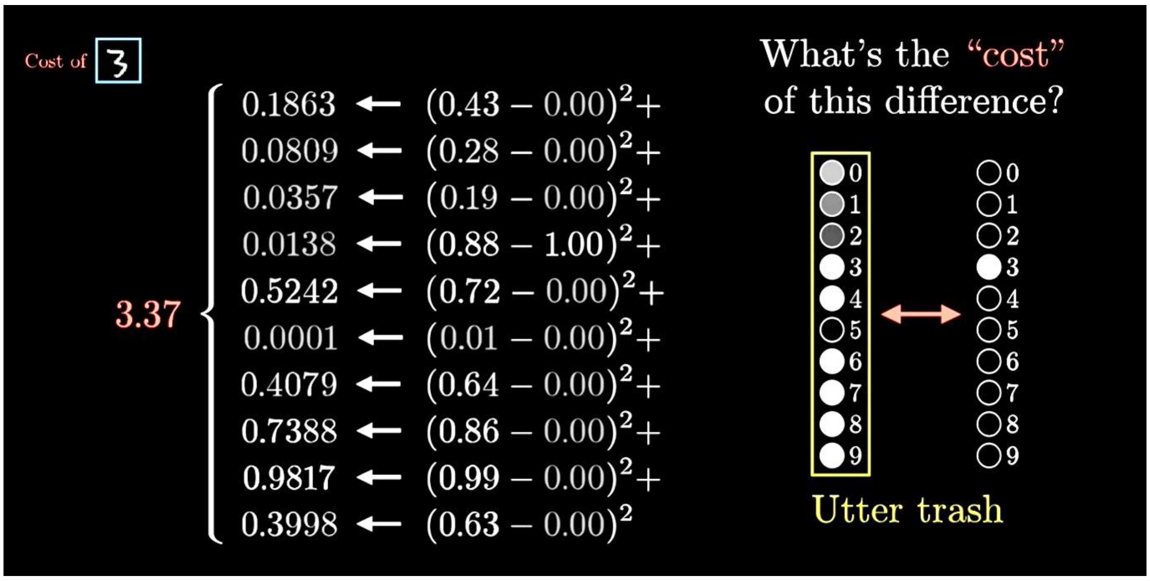


*Figure 28: Example of cost function value, in case of an untrained model*

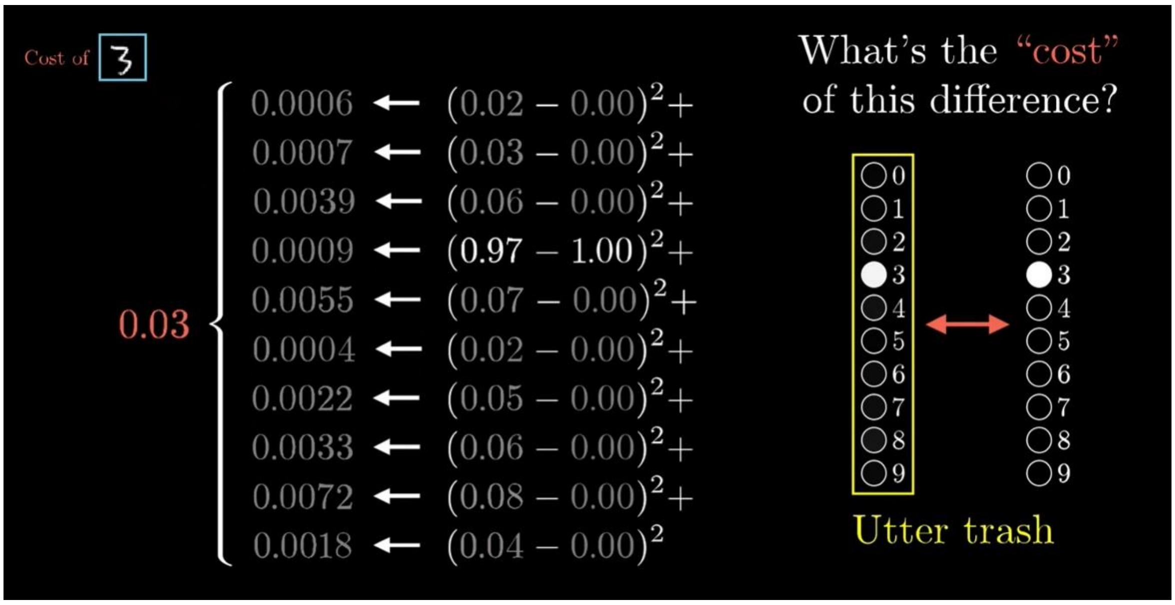
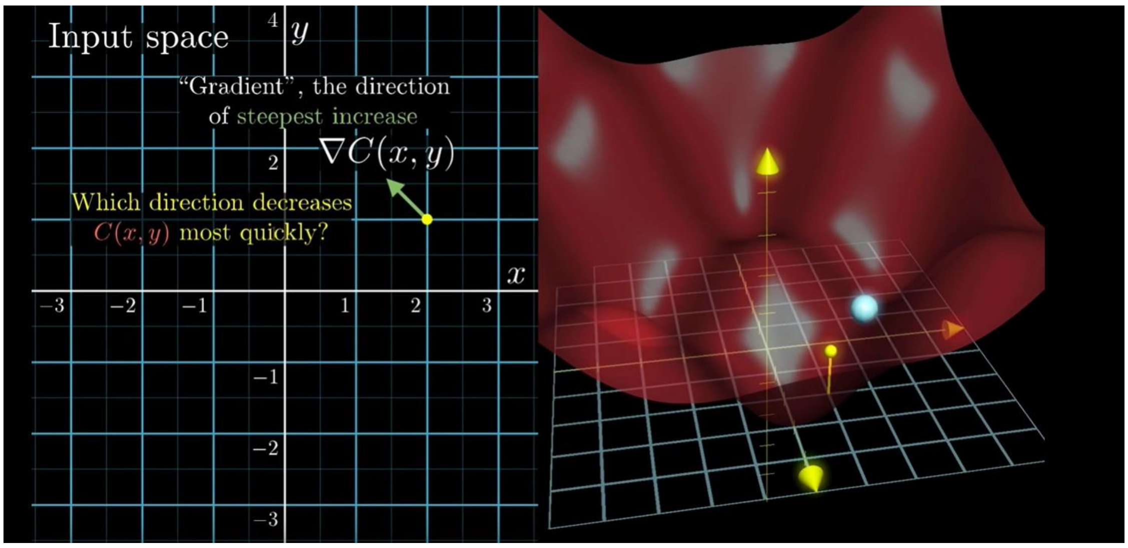


*Figure 29: Example for a well-trained model*

### 1. Gradient descent:

Finding the minima of a function that has thousands of parameters (these being the weights and biases), is pretty unimaginable by using partial derivatives, so to find the minimum we use what we call the gradient.

The gradient can be interpreted as the "direction and rate of fastest increase" from a certain point in the function, and by definition the negative of that gives the "direction and rate of fastest decrease" which eventually leads us to the minima of a given function.



Input space
$y$
"Gradient", the direction of steepest increase
$\nabla C(x, y)$
Which direction decreases $C(x, y)$ most quickly?
$x$

*Figure 30: Example of a gradient calculus for a function with two inputs*

Unlike the case for functions that have small numbers of inputs (or parameters), in ANN we use ***backpropagation*** algorithm to calculate the gradient.

Of course, since this algorithm is not our goal in this talk, we are not going to dive deep into details (you can check the link number **Link 1** for more). But the algorithm is pretty simple and intuitive, like illustrated in *Figure 31* below, what **backpropagation** does essentially is:

- For each data training sample;
- Going from right to left (from output layer towards input layer);
- Figuring out the change that should be made to each of the nodes connected to the objective node that we want to optimise its value, synapse by synapse (this is shown by the coloured arrows);
- Averaging these changes gives us the gradient related to this exact synapse (weight);

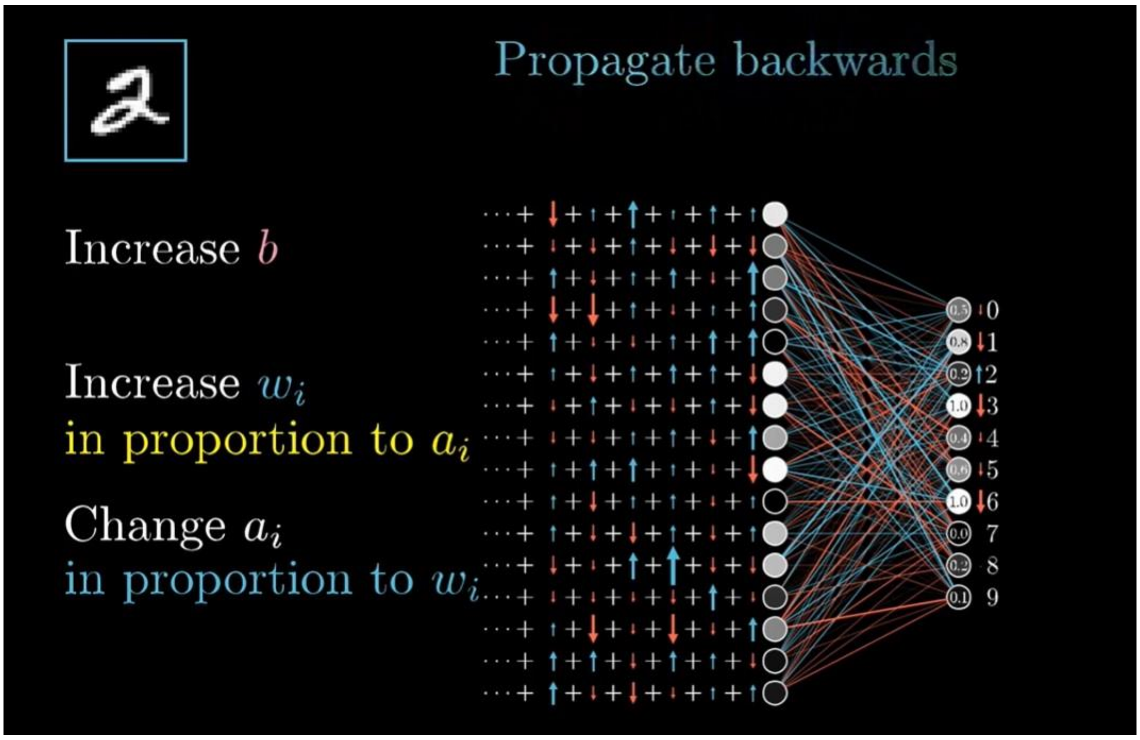


*Figure 31: Backpropagation algorithm simplified*

After one iteration of ***backpropagation*** we have a vector that has gradient of each single one parameter of the cost function, so we multiply this vector by the learning rate[1].

But careful, what we said a gradient does is give us the direction of the steepest increase, like for example in the *figure 32* below, we have a gradient equals 3.2 related to a weight 'A', and another gradient equals 0.1 related to a weight 'B', this means that a variation of weight 'A' is 33 times more effective and powerful than a variation in weight 'B'. Another example is shown in *figure 33*

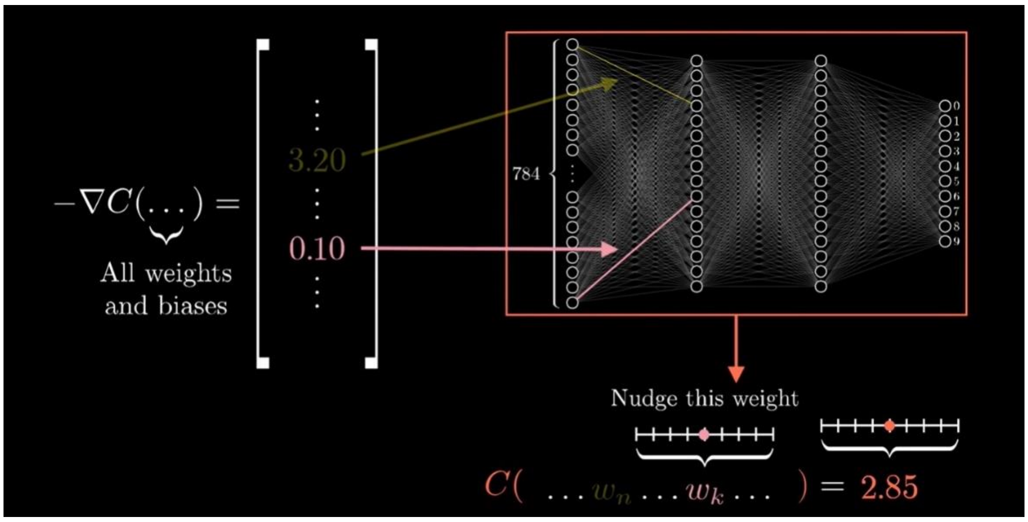
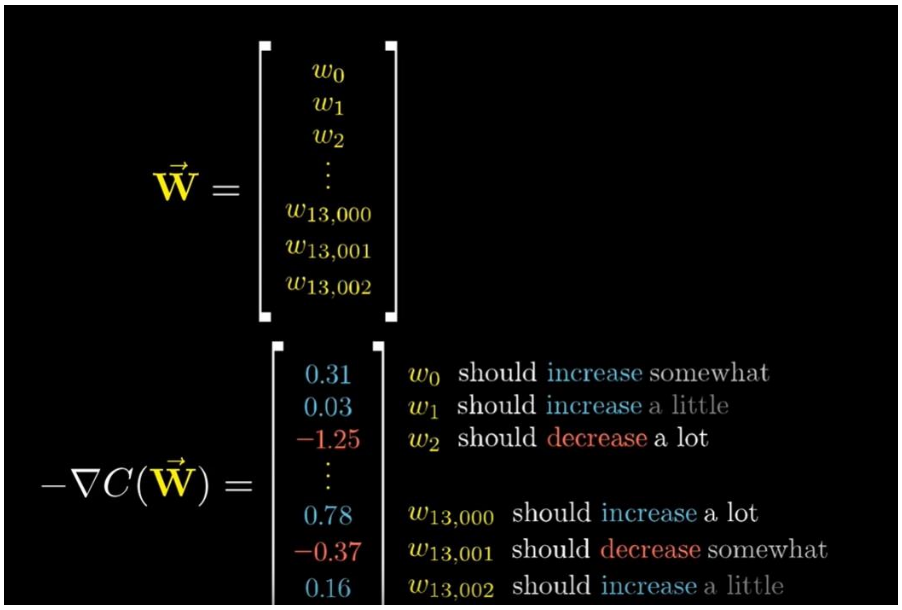


*Figure 32: Example of a gradient vector*



$\vec{\mathbf{W}} = [w_0, w_1, w_2, \vdots, w_{13,000}, w_{13,001}, w_{13,002}]$
$-\nabla C(\vec{\mathbf{W}}) = [0.31, 0.03, -1.25, \vdots, 0.78, -0.37, 0.16]$
$w_0$ should increase somewhat
$w_1$ should increase a little
$w_2$ should decrease a lot
$w_{13,000}$ should increase a lot
$w_{13,001}$ should decrease somewhat
$w_{13,002}$ should increase a little

*Figure 33: Example 2 of a gradient vector*

Now, after calculating the gradient of each weight and bias, we multiply it by the learning rate and we compute the loss function again to see the improvement.

Repeating this process to optimise the loss function, is what we call the **Gradient descent**.

[1] The step size of the learning rate is the amount that the weights are updated during training. Specifically, the learning rate is a configurable hyperparameter used in the training of neural networks that has a small positive value, often in the range between 0.0 and 1.0.

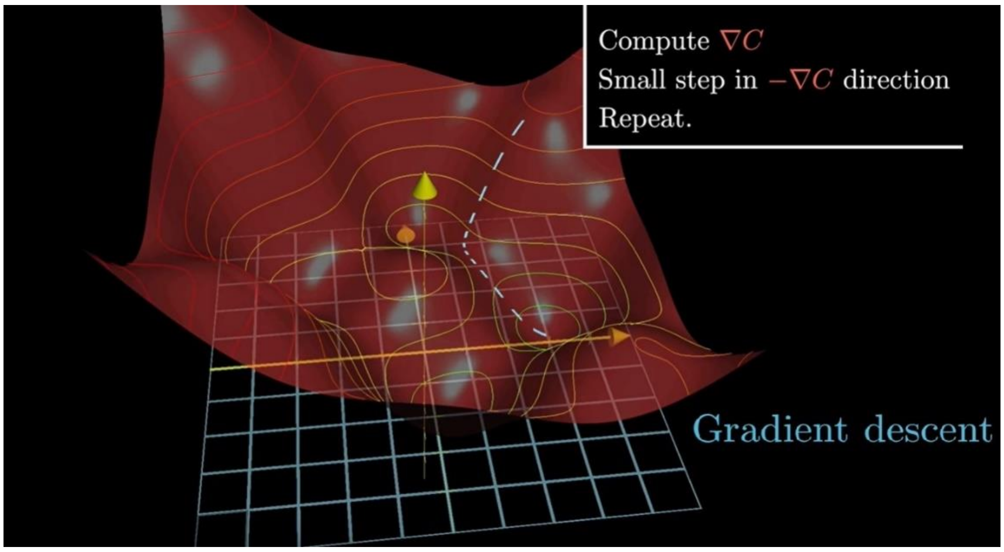


*Figure 34: Gradient descent, simplified*

## 2. Stochastic gradient descent:

To conclude, gradient descent basically is for each training data sample we:

- Compute the loss function
- Compute the gradient using **backpropagation** algorithm
- Multiplying the gradient by the learning step
- Tweaking the weights and biases to optimize the loss function
- Then Repeat the process again until we find the minima of the cost function

Well, clearly doing all of this for each data sample would make our learning very slow.

So in situations when we have large amounts of data, we use what we call **Stochastic gradient descent.**

In this variation, the gradient descent procedure described above is run but the update to the coefficients is performed for each training instance, rather than at the end of the batch of instances.

The first step of the procedure requires that the order of the training dataset is randomized (*Figure 35*). This is to mix up the order that updates are made to the coefficients. Because the coefficients are updated after every training instance, the updates will be noisy jumping all over the place, and so will the corresponding cost function. By mixing up the order for the updates to the coefficients, it harnesses this random walk and avoids it getting distracted or stuck.

The update procedure for the coefficients is the same as that above, except the cost is not summed over all training patterns, but instead calculated for one training pattern.

The learning can be much faster with stochastic gradient descent for very large training datasets and often you only need a small number of passes through the dataset to reach a good or good enough set of coefficients, e.g. 1-to-10 passes through the dataset.

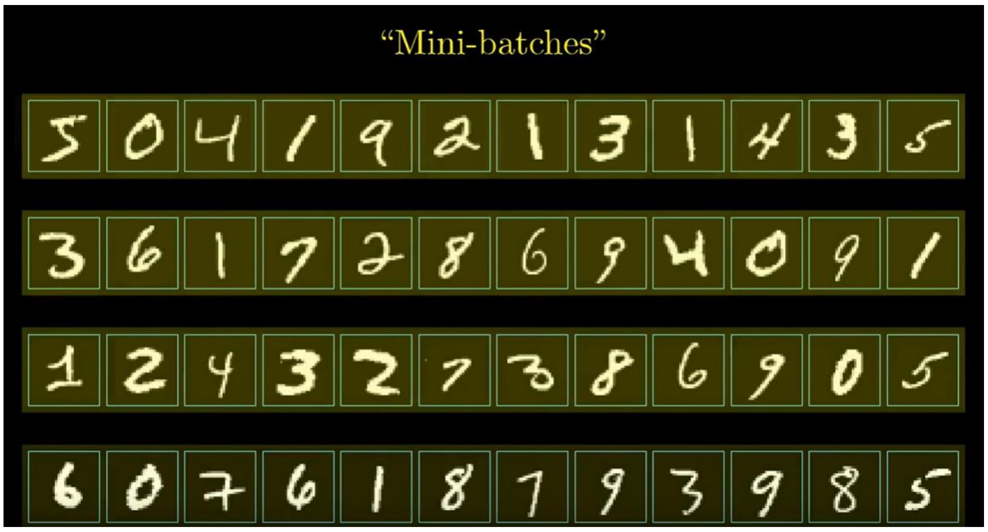


*Figure 35: Randomizing the data set and dividing it into small batches*

To illustrate the difference easily (*Figure 36*), we could compare, taking very calculated steps that insure us that we're going down towards the minima, against less calculated steps that points down to minima also, but not necessarily down the most optimized path.

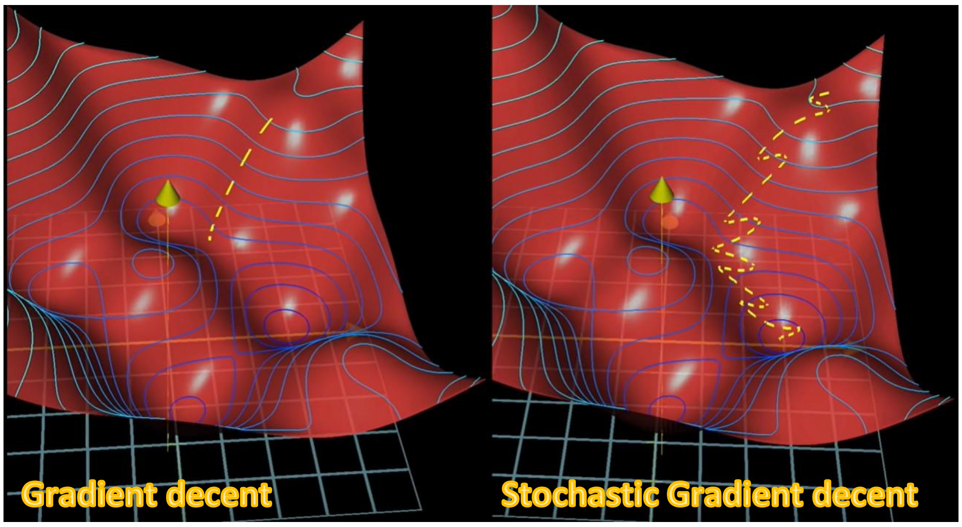


*Figure 36: Difference between stochastic gradient descent and gradient descent, Simplified*

## VI. Vanishing gradient problem:

### 1. Definition:

Vanishing Gradient Problem occurs when we try to train a Neural Network model using Gradient based optimization techniques (like the gradient descent or stochastic gradient descent).

What happens is that as we keep on adding more and more Hidden layers in the model, the learning speed of the next hidden layers in the model keep on getting slower and slower.

Generally, adding more hidden layers tends to make the network able to learn more complex arbitrary functions, and thus do a better job in predicting future outcomes. This is where Deep Learning is making a big difference due to the thousands and millions of hidden layers it has, we can now make sense of highly complicated data such as images, speeches, videos etc... and do Speech Recognition and Image Classification, Image Captioning etc.

### 2. Why does this problem occur?

Well certain activation functions, like the sigmoid function, squishes a large input space into a small input space between 0 and 1. Therefore, a large change in the input of the sigmoid function will cause a small change in the output. Hence, the derivative becomes small.

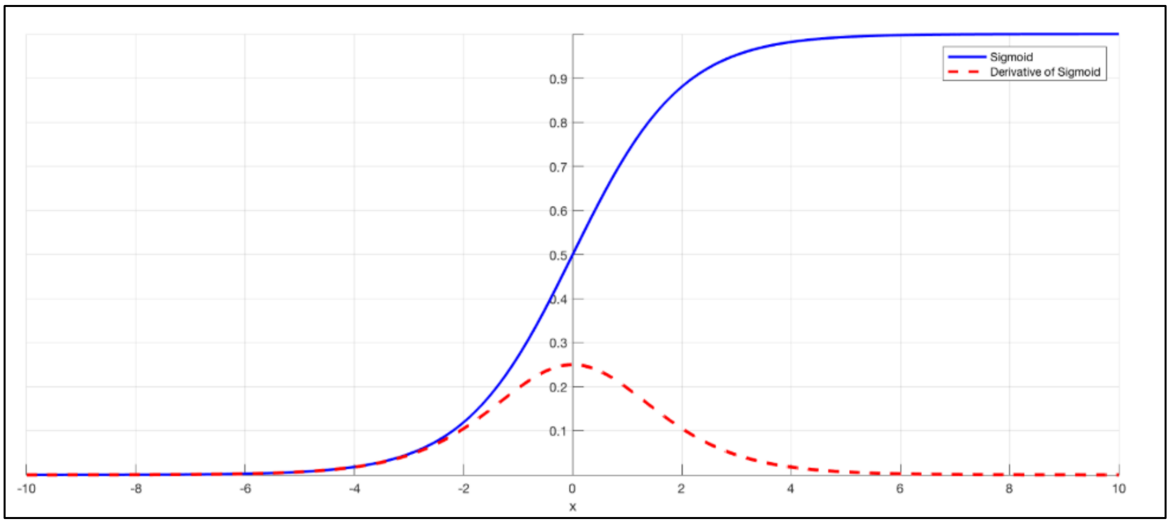


*Figure 37: The sigmoid function and its derivative*

As an example, (*Figure 37*) is the sigmoid function and its derivative. Note how when the inputs of the sigmoid function become larger or smaller (when |x| becomes bigger), the derivative becomes close to zero.

### 3. Why is it significant?

As we mentioned before, for shallow network with only a few layers that use these activations, this isn't a big problem. However, when more layers are used, it can cause the gradient to be too small for training to work effectively.

Gradients of neural networks are found using **backpropagation**. Simply put, **backpropagation** finds the derivatives of the network by moving layer by layer from the final layer to the initial one. By the chain rule[2], the derivatives of each layer are multiplied down the network (from the final layer to the initial) to compute the derivatives of the initial layers.

However, when n hidden layers use an activation like the sigmoid function, a number of small derivatives are multiplied together. The gradient decreases exponentially as we propagate down to the initial layers.

A small gradient means that the weights and biases of the initial layers will not be updated effectively with each training session. Since these initial layers are often crucial to recognizing the core elements of the input data, it can lead to overall inaccuracy of the whole network.

## 4. What solutions are there to prevent this problem?

The simplest solution is to use other activation functions, such as ReLU[3], which doesn't cause a small derivative.

Residual networks are another solution, as they provide residual connections straight to earlier layers. As seen in *Figure 38* below, the residual connection directly adds the value at the beginning of the block, x, to the end of the block (F(x)+x). This residual connection doesn't go through activation functions that "squashes" the derivatives, resulting in a higher overall derivative of the block.

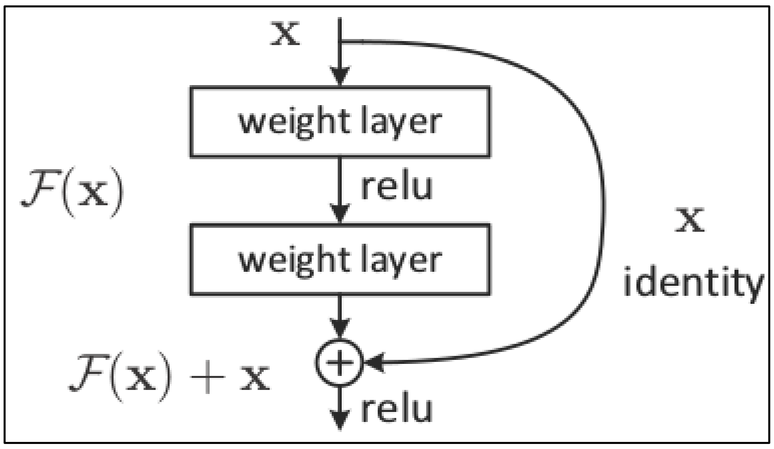

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

*Figure 38: Residual block*

Finally, batch normalization layers can also resolve the issue. As stated before, the problem arises when a large input space is mapped to a small one, causing the derivatives to disappear. In Image 1, this is most clearly seen at when |x| is big. Batch normalization reduces this problem by simply normalizing the input so |x| doesn't reach the outer edges of the sigmoid function. As seen in *Figure 39* bellow, it normalizes the input so that most of it falls in the green region, where the derivative isn't too small.

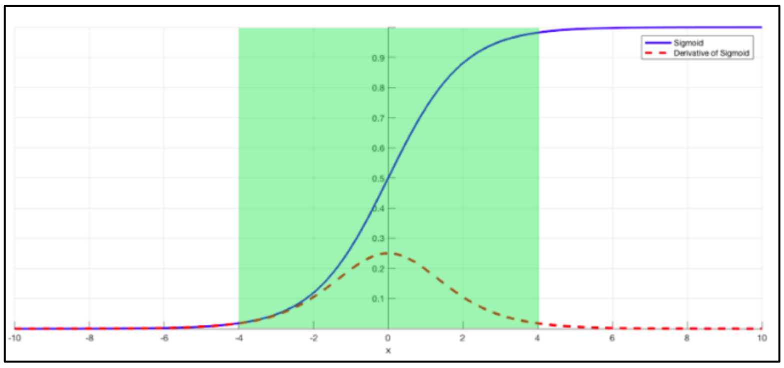


*Figure 39: Sigmoid function with restricted inputs*

[2] In calculus, the chain rule is a formula to compute the derivative of a composite function, it's what we use in calculus of backpropagation.

[3] ReLU: The Rectified Linear Unit activation function is a piecewise linear function that will output the input directly if is positive, otherwise, it will output zero.

# VII. APPLICATION

## 1. INTRODUCTION TO "PARKINSONS" DATASET

### 1.1. Parkinson disease

#### a. *Definition*

Parkinson's disease is a brain disorder that leads to shaking, stiffness, and difficulty with walking, balance, and coordination.

Parkinson's symptoms usually begin gradually and get worse over time. As the disease progresses, people may have difficulty walking and talking. They may also have mental and behavioral changes, sleep problems, depression, memory difficulties, and fatigue.

Both men and women can have Parkinson's disease. However, the disease affects about 50 percent more men than women.

#### b. *What Causes Parkinson's Disease?*

Parkinson's disease occurs when nerve cells, or neurons, in an area of the brain that controls movement become impaired and/or die. Normally, these neurons produce an important brain chemical known as dopamine. When the neurons die or become impaired, they produce less dopamine, which causes the movement problems of Parkinson's. Scientists still do not know what causes cells that produce dopamine to die.

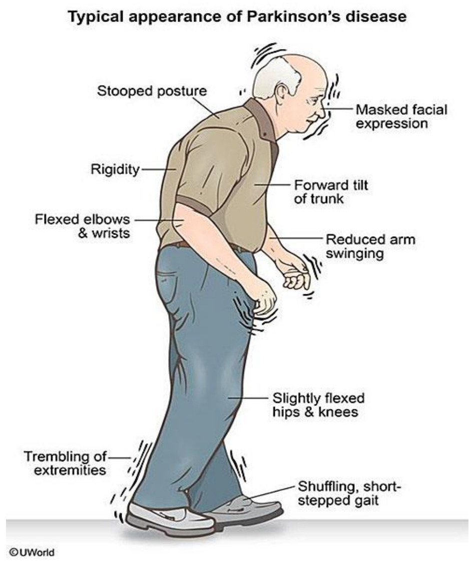


*Figure 40: Typical Appearance of Parkinson disease*

#### c. *Symptoms of Parkinson's Disease*

Parkinson's disease has four main symptoms:

- Tremor (trembling) in hands, arms, legs, jaw, or head
- Stiffness of the limbs and trunk
- Slowness of movement
- Impaired balance and coordination, sometimes leading to falls

### 1.2. Parkinson disease and machine learning

Parkinson's disease (PD) patient care is limited by inadequate, sporadic symptom monitoring, infrequent access to care, and sparse encounters with healthcare professionals leading to poor medical decision making and sub-optimal patient health-related outcomes. More frequent patient monitoring and treatment adjustments can lead to better symptomatic management and reduction in treatment-related complications such as motor fluctuations.

Recent advances in digital health approaches have enabled objective and remote monitoring of impaired motor function with the promise of profoundly changing the diagnostic, monitoring, and therapeutic landscape in PD. Sensing technologies, mobile networks, cloud computing, the Internet of Things and big data analytics innovations that have the potential to transform healthcare and our approach to patients with chronic, complex, and disorders like PD.

## 1.3. Parkinsons Disease Data Set

*The dataset for the disease is retrieved from UCI repository (**http://archive.ics.uci.edu/ml/datasets/parkinsons**).*

*a.* *Source:*

The dataset was created by **Max Little** of the University of Oxford, in collaboration with the **National Centre for Voice and Speech**, Denver, Colorado, who recorded the speech signals. The original study published the feature extraction methods for general voice disorders.

*b.* *Data Set Information:*

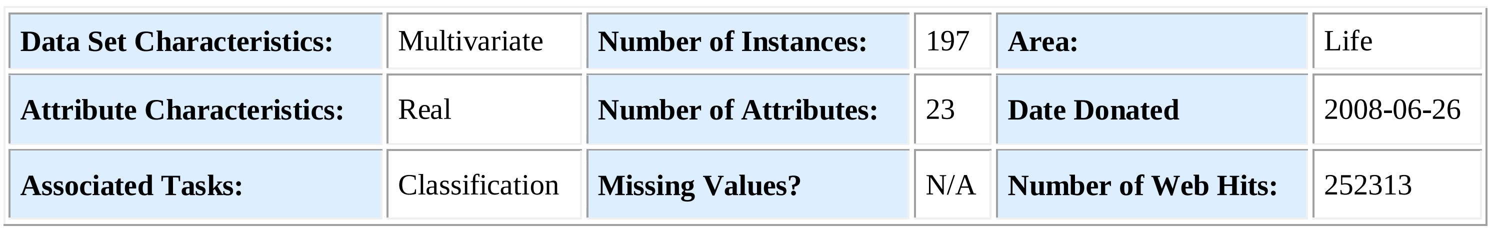
This dataset is composed of a range of biomedical voice measurements from 31 people, 23 with Parkinson's disease (PD). Each column in the table is a particular voice measure, and each row corresponds one of **195 voice recordings** from these individuals ("name" column). The main aim of the data is to discriminate healthy people from those with PD, according to "**status**" column which is set to 0 for healthy and 1 for PD.

The data is in ASCII CSV format. The rows of the CSV file contain an instance corresponding to one voice recording. There are around six recordings per patient, the name of the patient is identified in the first column.

*c.* *Attribute Information:*

Matrix column entries (attributes):

- Name - ASCII subject name and recording number
- MDVP: Fo(Hz) - Average vocal fundamental frequency
- MDVP: Fhi(Hz) - Maximum vocal fundamental frequency
- MDVP: Flo(Hz) - Minimum vocal fundamental frequency
- MDVP: Jitter(%), MDVP:Jitter(Abs),MDVP:RAP,MDVP:PPQ,Jitter:DDP – Several measures of variation in fundamental frequency
- MDVP: Shimmer,MDVP:Shimmer(dB),Shimmer:APQ3,Shimmer:APQ5,MDVP:APQ,Shimmer: DDA - Several measures of variation in amplitude
- NHR, HNR - Two measures of ratio of noise to tonal components in the voice
- Status - Health status of the subject (one) - Parkinson's, (zero) - healthy
- RPDE, D2 - Two nonlinear dynamical complexity measures
- DFA - Signal fractal scaling exponent
- spread1, spread2, PPE - Three nonlinear measures of fundamental frequency variation



| | | | | | |
|---|---|---|---|---|---|
| **Data Set Characteristics:** | Multivariate | **Number of Instances:** | 197 | **Area:** | Life |
| **Attribute Characteristics:** | Real | **Number of Attributes:** | 23 | **Date Donated** | 2008-06-26 |
| **Associated Tasks:** | Classification | **Missing Values?** | N/A | **Number of Web Hits:** | 252313 |

*Figure 41: summary table of the characteristics of the PD dataset*

### 1.4. The approach of the project

A relative study on feature relevance analysis and the accuracy using different classification methods was carried out on Parkinson dataset.

Computer-aided-diagnosis systems based on machine learning can be useful in assisting clinicians in identifying PD patients.

In this project, we evaluate the performance of machine learning based techniques for PD diagnosis, based specifically on **ARTIFICIAL NEUROL NETWORKS** model (ANN).

This machine learning method included the **keras** classifier, the **sequential** model the **Stochastic gradient descent** optimizer, and many other used libraries, functions and methods. We evaluated the performance of this method by means of a train/test/split validation protocol.

## 2. TOOLS AND METHODS USED

### 2.1. Language and software

In this project we worked with **Python** coding language, and we build our test application in **Jupyter** notebook, with the **anaconda** software as a launcher.

;

a. *Python programming Language*

**Python** is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991. Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects. Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly, procedural,) object-oriented, and functional programming. Python is often described as a "batteries included" language due to its comprehensive standard library.

b. *Anaconda Software*

**Anaconda** is a free and open-source distribution of the Python and R programming languages for scientific computing (data science, machine learning applications, large-scale data processing, predictive analytics, etc.), that aims to simplify package management and deployment. Package versions are managed by the package management system conda.

c. *Jupyter Notebook*

**Jupyter** Notebook (formerly IPython Notebooks) is a web-based interactive computational environment for creating Jupyter notebook documents. The "notebook" term can colloquially make reference to many different entities, mainly the Jupyter web application, Jupyter Python web server, or Jupyter document format depending on context. A Jupyter Notebook document is a JSON document, following a versioned schema, and containing an ordered list of input/output cells which can contain code, text (using Markdown), mathematics, plots and rich media, usually ending with the ".ipynb" extension.

## 2.2. Libraries and modules

*a.* *Pandas*

In computer programming, **Pandas** is a software library written for the Python programming language for data manipulation and analysis.It’s a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

*b.* *Sklearn*

**Sklearn** (scikit-learn) is a Python module integrating classical machine learning algorithms in the tightly-knit world of scientific Python packages (numpy, scipy, matplotlib).
It aims to provide simple and efficient solutions to learning problems that are accessible to everybody and reusable in various contexts: machine-learning as a versatile tool for science and engineering.

*c.* *Keras*

**Keras** is a high-level neural networks API, written in Python and capable of running on top of **TensorFlow**, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. It has the capability to go from idea to result with the least possible delay.

It allows also to define and train neural network models in just a few lines of code.

*d.* *Matplotlib*

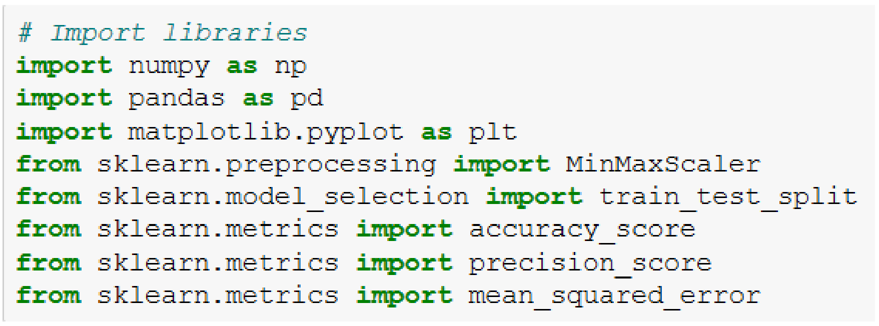
**Matplotlib** is a comprehensive library for creating static, animated, and interactive visualizations in Python.

## 3. IMPLEMENTATION

In order to implement the application, we proceeded the following steps:

- Import libraries
- Load Data and process it
- Define the Keras Model and compile it
- Fit the Keras Model
- Make Predictions
- Evaluate the Keras Model

## 3.1. Importing main libraries:

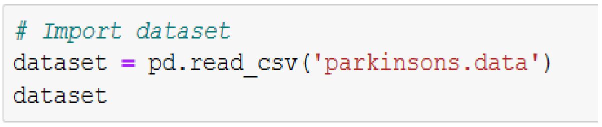


```python
# Import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import mean_squared_error
```

*Figure 42: Importing Libraries code*

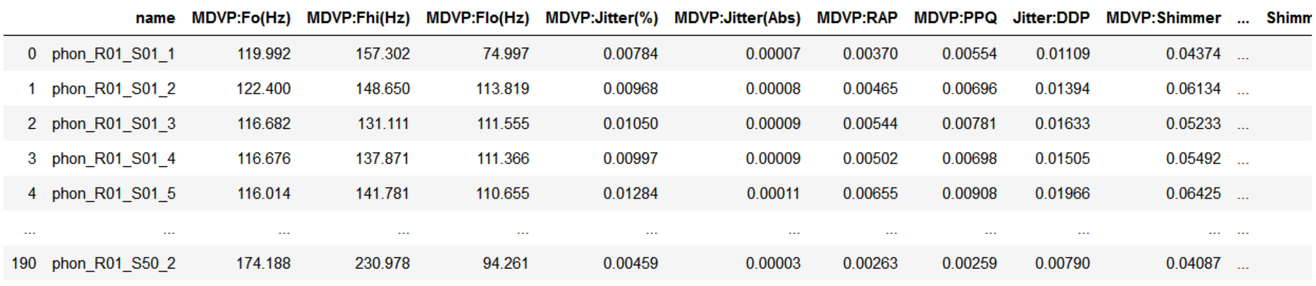
## 3.2. Load Data and process it

### a. *Load Data*



```python
# Import dataset
dataset = pd.read_csv('parkinsons.data')
dataset
```

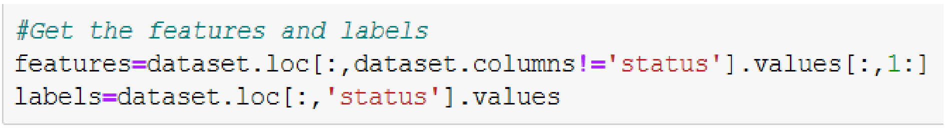
We import the dataset already downloaded in the project file by using the Pandas library defined previously as ‘pd’, and the reads method.
After that we display the dataset, and as we can see, the dataset corresponds to the description.



```
           name  MDVP:Fo(Hz)  MDVP:Fhi(Hz)  MDVP:Flo(Hz)  MDVP:Jitter(%)  MDVP:Jitter(Abs)  MDVP:RAP  MDVP:PPQ  Jitter:DDP  MDVP:Shimmer  ...  Shimm
  0  phon_R01_S01_1      119.992       157.302        74.997         0.00784           0.00007   0.00370   0.00554     0.01109       0.04374  ...
  1  phon_R01_S01_2      122.400       148.650       113.819         0.00968           0.00008   0.00465   0.00696     0.01394       0.06134  ...
  2  phon_R01_S01_3      116.682       131.111       111.555         0.01050           0.00009   0.00544   0.00781     0.01633       0.05233  ...
  3  phon_R01_S01_4      116.676       137.871       111.366         0.00997           0.00009   0.00502   0.00698     0.01505       0.05492  ...
  4  phon_R01_S01_5      116.014       141.781       110.655         0.01284           0.00011   0.00655   0.00908     0.01966       0.06425  ...
...             ...          ...           ...           ...             ...               ...       ...       ...         ...           ...  ...
190  phon_R01_S50_2      174.188       230.978        94.261         0.00459           0.00003   0.00263   0.00259     0.00790       0.04087  ...
```

*Figure 43: Partial View of the Parkinson Dataset Output*

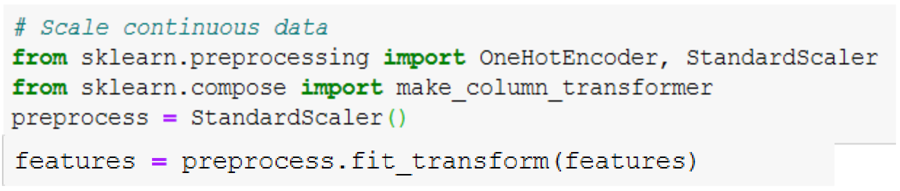
### b. *Process Data*



```python
#Get the features and labels
features=dataset.loc[:,dataset.columns!='status'].values[:,1:]
labels=dataset.loc[:,'status'].values
```

*Figure 44: Part 1 of data Processing code*

After loading the data, we get the features and labels:
- Features are extracted from the dataset by combining all the dataset values and excluding the ‘status’ column, which is our main attribute to predict in this application.
- Labels on the other hand are all the rows of the dataset without any columns besides the one with the title ‘status’.



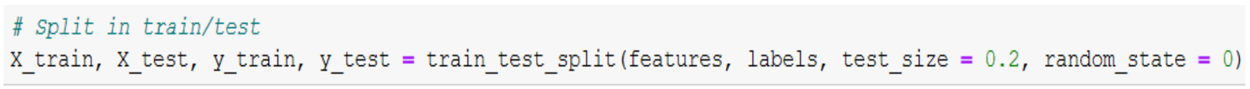
```python
# Scale continuous data
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import make_column_transformer
preprocess = StandardScaler()

features = preprocess.fit_transform(features)
```

*Figure 45: Part 2 of data Processing code*

The second step of data processing is preprocessing it by calling the **Standard Scaler** Class of the Sklearn.preprocessing package in order to standardize the data, in case outliers are present in the set.

After that we apply the **fit_transfom()** method which is simply the **fit()** method followed by the **transform()** method ,so by fit the imputer calculates the means of columns from some data, and by transform it applies those means to some data (which is just replacing missing values with the means).



```
# Split in train/test
X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size = 0.2, random_state = 0)
```

*Figure 46: Part 3 of data Processing code*

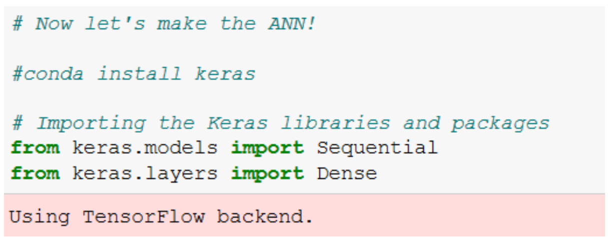
Lastly, we split the dataset into a **train set** and a **test set** with the **train_test_split()** method from the sklearn.model_selection package ,with a test size of 0.2 so that : **<u>Train dataset = 80 %</u>** | **<u>Test dataset = 20%</u>**

## 3.3. Define Keras Model and compile it

### a. *<u>Define Keras Model</u>*

After loading the data and processing it we mode to defining the keras model which is the artificial neurol network
Before proceeding as mentioned, some keras libraries and packages are needed to be imported and installed

- The keras library is installed in the anaconda software prompt.
- And the keras '**Sequential'** model and '**Dense'** class are imported from the keras library



```
# Now let's make the ANN!

#conda install keras

# Importing the Keras libraries and packages
from keras.models import Sequential
from keras.layers import Dense
```

```
Using TensorFlow backend.
```

*Figure 47: Keras Libraries Importation code*

We will define our neurol network in a customized function called **build_classifier**

Models in Keras are defined as a sequence of layers.

- We begin by creating a **Sequential model** and adding layers one at a time until we are happy with our network architecture.

- The first thing to get right is to ensure the input layer has the right number of input features. This can be specified when creating the first layer with the **input_dim** argument and setting it to **22** for the 22 input variables.

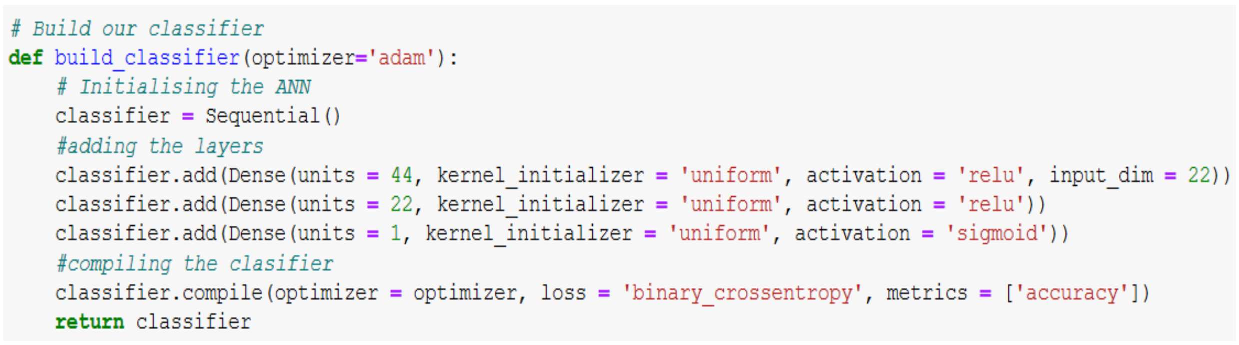
- Generally, we need a network large enough to capture the structure of the problem. In this application, we will use a fully-connected network structure with **3 layers**.

- Fully connected layers are defined using the **Dense class**. We can specify the number of neurons or nodes in the layer as the first argument and specify the activation function using the **activation** argument.

- We will use the **rectified linear unit activation function** referred to as **ReLU** on the first two layers and the **Sigmoid** function in the output layer.
Better performance is achieved using the ReLU activation function. We use a sigmoid on the output layer to ensure our network output is between 0 and 1.

We can summarize it all together by adding each layer:

- The model expects rows of data with 22 variables (the ***input_dim=22*** argument)
- The first hidden layer has **44** nodes and uses the relu activation function.
- The second hidden layer has **22** nodes and uses the relu activation function.
- The output layer has **1** node and uses the sigmoid activation function.



```
# Build our classifier
def build_classifier(optimizer='adam'):
    # Initialising the ANN
    classifier = Sequential()
    #adding the layers
    classifier.add(Dense(units = 44, kernel_initializer = 'uniform', activation = 'relu', input_dim = 22))
    classifier.add(Dense(units = 22, kernel_initializer = 'uniform', activation = 'relu'))
    classifier.add(Dense(units = 1, kernel_initializer = 'uniform', activation = 'sigmoid'))
    #compiling the clasifier
    classifier.compile(optimizer = optimizer, loss = 'binary_crossentropy', metrics = ['accuracy'])
    return classifier
```

*Figure 48: Defining the Keras model code*

*b.* *Compile the keras model*

Now that the model is defined, *we can compile it.*

Compiling the model uses the efficient numerical libraries under the covers (backend) such as Theano or **TensorFlow**.

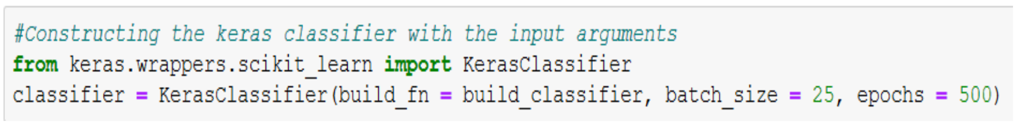
We must specify the loss function to use to evaluate a set of weights, the optimizer is used to search through different weights for the network and any optional metrics we would like to collect and report during training.

In this case, we will use cross entropy as the **loss** argument. In our case this loss is for a binary classification problem and is defined in Keras as **binary_crossentropy**.

We will define the **optimizer** as the efficient **stochastic gradient descent** algorithm "**adam** ".

**"ADAM"** is an adaptive learning rate optimization algorithm that's been designed specifically for training deep neural networks. It's a popular version of gradient descent because it automatically tunes itself and gives good results in a wide range of problems.

Finally, because it is a classification problem, we will collect and report the classification **accuracy**, defined via the **metrics** argument.



```python
#Constructing the keras classifier with the input arguments
from keras.wrappers.scikit_learn import KerasClassifier
classifier = KerasClassifier(build_fn = build_classifier, batch_size = 25, epochs = 500)
```

*Figure 49: Compiling the Keras model code*

The KerasClassifier classes in Keras take an argument **build_fn** which is the name of the function to call to get the model.

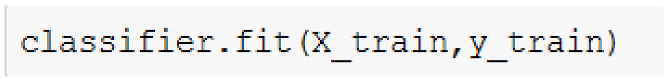
We already defined a function called **build_classifier** that defines our model, compiles it and returns it.

We pass this function name to the KerasClassifier class by the **build_fn** argument as shown in the figure 14 . We also pass in additional arguments:

**nb_epoch=500** | **batch_size=25**.

## 3.4. Fit Keras Model

We have defined our model and compiled it and it's ready for efficient computation.

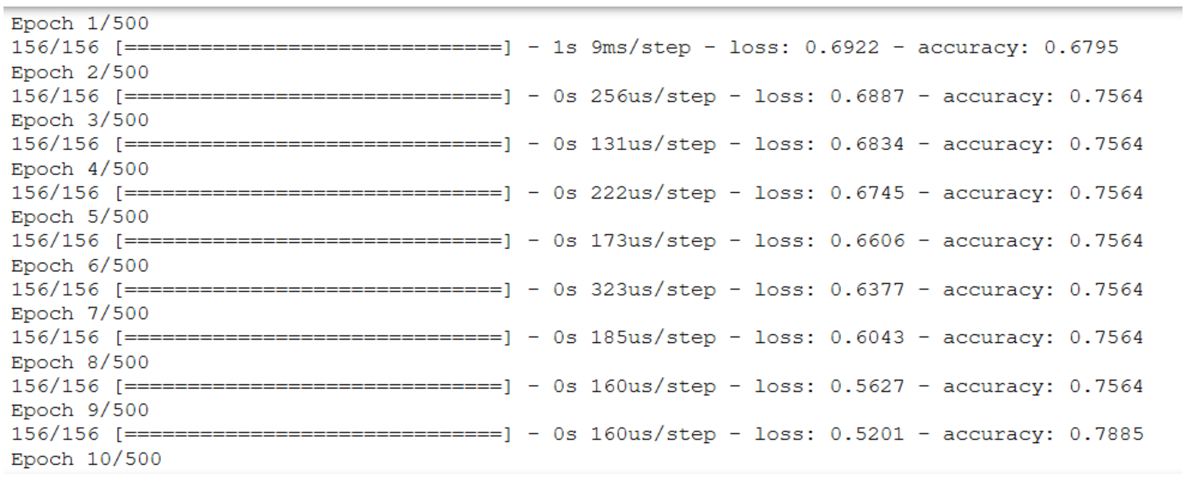


```python
classifier.fit(X_train,y_train)
```

Now it is time to execute the model on some data.
We will train or fit our model on the portion of the data of training (**X_train, y_train**) by calling the **fit()** function on the model.
Training occurs over epochs and each epoch is split into batches.

**Epoch**: One pass through all of the rows in the training dataset.
**Batch**: One or more samples considered by the model within an epoch before weights are updated.

The output is as follows:



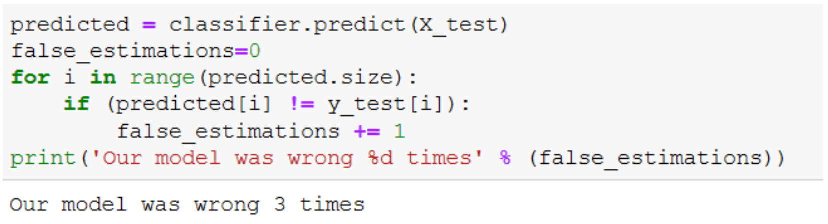
```
Epoch 1/500
156/156 [==============================] - 1s 9ms/step - loss: 0.6922 - accuracy: 0.6795
Epoch 2/500
156/156 [==============================] - 0s 256us/step - loss: 0.6887 - accuracy: 0.7564
Epoch 3/500
156/156 [==============================] - 0s 131us/step - loss: 0.6834 - accuracy: 0.7564
Epoch 4/500
156/156 [==============================] - 0s 222us/step - loss: 0.6745 - accuracy: 0.7564
Epoch 5/500
156/156 [==============================] - 0s 173us/step - loss: 0.6606 - accuracy: 0.7564
Epoch 6/500
156/156 [==============================] - 0s 323us/step - loss: 0.6377 - accuracy: 0.7564
Epoch 7/500
156/156 [==============================] - 0s 185us/step - loss: 0.6043 - accuracy: 0.7564
Epoch 8/500
156/156 [==============================] - 0s 160us/step - loss: 0.5627 - accuracy: 0.7564
Epoch 9/500
156/156 [==============================] - 0s 160us/step - loss: 0.5201 - accuracy: 0.7885
Epoch 10/500
```

*Figure 50: Output of model training*

As we can see the model is training on the data through each one of the 500 epochs and recording the accuracy and the loss values upon each iteration.

## 3.5. Make Predictions

In order to make a preliminary evaluation of the performance of our model, we apply a manual test on the predictions of the model for the **test dataset** (X_test, y_test)



```
predicted = classifier.predict(X_test)
false_estimations=0
for i in range(predicted.size):
    if (predicted[i] != y_test[i]):
        false_estimations += 1
print('Our model was wrong %d times' % (false_estimations))
```

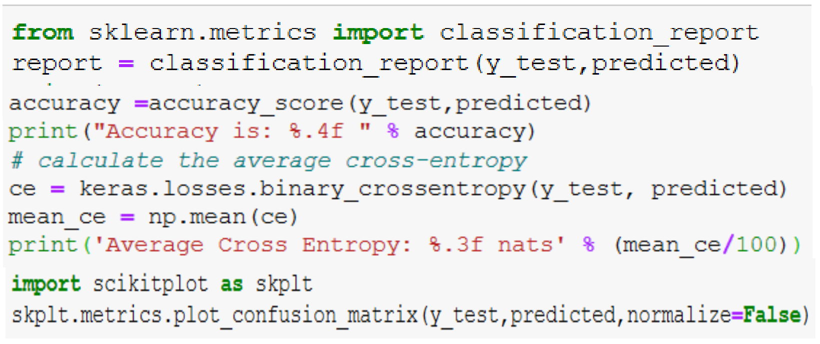
```
Our model was wrong 3 times
```

*Figure 51: Making prediction code*

## 3.6. Evaluate the Keras Model

In order to evaluate the efficiency of our model, we incorporated and used multiple tools and metrics:

- Classification report
- Accuracy Score
- Binary Cross Entropy loss
- Confusion Matrix
- Roc Curve and roc_auc_score



```
from sklearn.metrics import classification_report
report = classification_report(y_test,predicted)
accuracy =accuracy_score(y_test,predicted)
print("Accuracy is: %.4f " % accuracy)
# calculate the average cross-entropy
ce = keras.losses.binary_crossentropy(y_test, predicted)
mean_ce = np.mean(ce)
print('Average Cross Entropy: %.3f nats' % (mean_ce/100))
import scikitplot as skplt
skplt.metrics.plot_confusion_matrix(y_test,predicted,normalize=False)
```

*Figure 52: Metrics code*

### a. *Metrics*

#### i. *Classification report*

The classification report displays the precision, recall, F1, and support scores for the model. The metrics are defined in terms of true and false positives, and true and false negatives

**Precision**: The ability of a classifier not to label an instance positive that is actually negative.
**Recall**: The ability of a classifier to find all positive instances.
**f1 score**: A weighted harmonic mean of precision and recall such that the best score is 1.0 and the worst is 0.0.
**support**: Support is the number of actual occurrences of the class in the specified dataset.

#### ii. *Accuracy Score*

The number of correct made predictions as a ratio of all predictions made.

#### iii. *Binary Cross Entropy*

The loss function used in Classification problems are those that involve one or more input variables and the prediction of a class label.

Binary Cross entropy is used in classification tasks that have just two labels for the output variable.

*iv. Confusion Matrix*

A handy presentation of the accuracy of a model with two or more classes

*v. Roc Curve and roc auc score*

Area Under **ROC Curve** (or ROC AUC for short) is a performance metric for binary classification problems. The AUC represents a model's ability to discriminate between **positive** and **negative** classes. An area of **1.0** represents a model that made all predictions perfectly. An area of **0.5** represents a model as good as random.

b. *Results of the metrics*

*i. The Classification Report*

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.82 | 0.90 | 0.86 | 10 |
| 1 | 0.96 | 0.93 | 0.95 | 29 |
| Accuracy | | | 0.92 | 39 |
| Macro avg | 0.89 | 0.92 | 0.90 | 39 |
| Weighted avg | 0.93 | 0.92 | 0.92 | 39 |

*Figure 53: Table of the classification report*

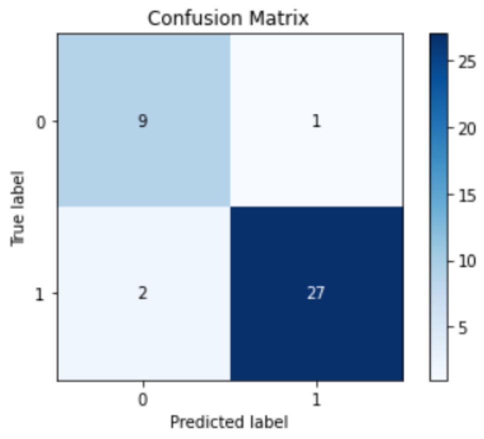
*ii. The Accuracy Score*

The model recorded an accuracy Value of: **0.9231 ≈ 92%**

*iii. Binary Cross Entropy*

The model recorded an Cross entropy loss mean Value of: **0.061**

*iv. Confusion Matrix*



From the confusion matrix plot:

- True Positives = 27
- True Negatives = 9
- False Positives = 2
- False Negative = 1

*Figure 54: Plot of the confusion matrix*

*v. ROC Curve and roc auc score*

The model recorded a roc auc score Value of: **0.959**

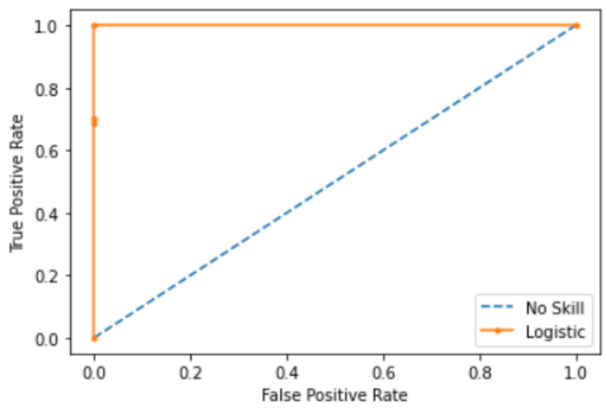


*Figure 55: ROC Curve plot*

## c. *Training Process*

In order to make a deep study on the efficiency of the model, we made a follow up of the training process of the classifier, and constructed 2 plots of the accuracy and loss values:

- The first Plot: training accuracy values
- The second plot: training loss values

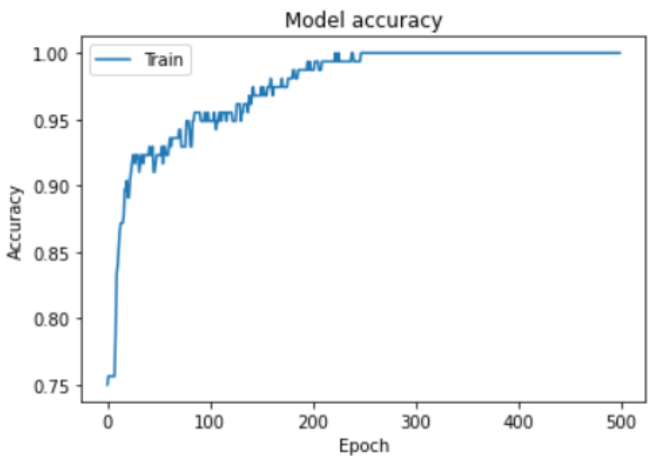


*Figure 56: training accuracy values graph plot*

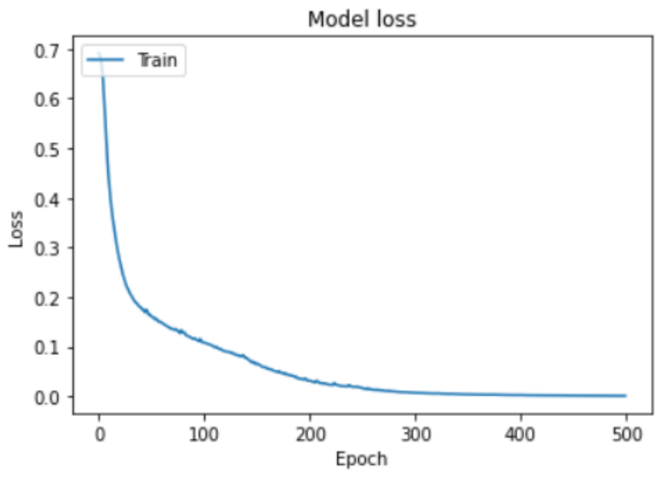


*Figure 57: training loss values graph plot*

## 4. RESULTS INTERPRETATION

In the evaluation process of our model, we used many classification evaluation metrics.
This will only give us an idea of how well we have modeled the dataset, but no idea of how well the algorithm might perform on new data.
In order to avoid simplicity and adopt an ideal evaluation process, we have separated our data into train and test datasets (figure 11) for training and evaluation of our model.

For a preliminary evaluation, we have made predictions of the test dataset (20% of the global dataset) and the results were as follows:

| Test Dataset | True Predictions | Wrong predictions |
|---|---|---|
| **39** | **36** | **3** |

*Figure 58: Predictions results on the test dataset*

As we can see, out of 20% test dataset (39 voice recordings = 39 dataset rows), we only got 3 wrong predictions

### 4.1. Training Process interpretation

From both plots, in Figure 22 and figure 23, we can say that the model is efficient enough since throughout the training process, the accuracy and the loss values become better upon each iteration :

***Accuracy value Rising*** ***&*** ***Loss Value dropping***

### 4.2. Classification metrics Interpretation

a. *Classification Report*

The classification report in figure 19 shows that 98% of the predictions of the first class and 82% of the predictions of the second class are actually of the predicted class (precision).
In addiction, the model is correctly identifying 90% % of the class 0s, and 93% of the class 1s (recall).

b. *The Accuracy Score*

Accuracy is the most intuitive performance measure and it is simply a ratio of correctly predicted observation to the total observations. We have got 0.9231 which means our model is approximately **92%** accurate.

c. *Binary Cross Entropy*

The model recorded a mean of losses of approximately **0.06** which means our model is efficient and makes good predictions

d. *Confusion Matrix*

In the confusion matrix in figure 20, the value of true positives is very high compared to the false positives and the same I applied for the true negatives

e. *ROC Curve and roc auc score*

From the plot in figure 21, it appears that the model is indeed efficient since the AUC (area under curve) of the plot is wide and the true positive rate approximately equals 1

On the other hand, the roc auc score reaches **0.959**

Experimental results show that the ANN based keras classifier achieved a high average performance in terms of overall accuracy, precision, confusion matrix results and area under the curve plot. Predictions were made to evaluate the model’s efficiency on new data, confirming that the model outperformed in terms of metrics measures. The artificial neurol network based keras model is a promising method for identifying PD patients based on binary classification.

# CONCLUSION

ANNs are considered as simple mathematical models to enhance existing data analysis technologies. Although it is not comparable with the power of the human brain, still it is the basic building block of the Artificial intelligence. Our research shows an example that could be implemented in the medical field, but despite of the respectable accuracy we've got as a result, there is still space for improvements. This project served us a small introduction to AI technology, one of the most important technologies to learn by students, not only because it's improving day by day, it's also because it's now considered the new evolution in science.

# LISTE OF FIGURES :

# Bibliography

## Article Referred:

**A**.Krizhevsky, **G**.Hinton, **I**.Sutskever, **N**.Srivastava and **R**.Salakhutdinov.Editor **Y**.Bengio *Dropout: A Simple Way to Prevent Neural Networks from Overfitting 1930-1936, 2014*

## Site Referred:

http://archive.ics.uci.edu/ml/datasets/parkinsons ,dataset

https://github.com/mohamedOubella/GSEII_2_AI_Project?fbclid=IwAR3YoaGvRJoCHQqOeAfcQ_t2rtkCKbfJDmje0aVaR_GhJAkQyQBqARIKiF0 ,Application

https://pandas.pydata.org/

https://kite.com/python/docs/sklearn

https://matplotlib.org/

https://machinelearningmastery.com/

https://www.scikityb.org/en/latest/api/classifier/

https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

http://neuralnetworksanddeeplearning.com/

https://machinelearningmastery.com/gradient-descent-for-machine-learning/