What are the performances and limits of a perceptron in Artificial Neural Networks?

Reference to sources are made as follows: [n°] where n° is the number of the source which can be found in the bibliography

Introduction

Artificial Neural Networks is likely to be one of the most significant inventions of this century as it could soon mark the beginning of a new era, Artificial Intelligence.
The Neural Network is at the forefront of AI which will most likely revolutionize many industries such as biomedical industry, physics, finance, big data, multimedia security and many others. I knew that Neural networks were already used in some programs, such as image recognition (face id, detecting an object in an image), but also in handwriting recognition and many other applications. Neural networks offer an infinite amount of possible applications in so many fields.
As a computer science student and tech-enthusiast, neural networks is one of my most passionate topics and I've been wanting to study this subject in detail for a while. Which is the reason why I picked it for my IA.
In this research I will explore some of the maths behind Artificial Neural Networks. Such as probabilities, derivative, function analysis, mathematical analysis, which will be discussed in this study.

Description

What is a Artificial neural Network (ANN):

Introduce the global problem and analogy with biological neural networks

Imagine that you had the task of writing a program that has to identify a handwritten number, the human brain is capable of accomplishing this task very easily, it uses its biological neural network to process the visual input information. It does so by sending electrical signals between the interconnected neurons until it outputs the associated idea.

Introducing artificial neural network

So in order to resolve this problem, humanity recreated an artificial neural network (ANN) that mimics the human brain's way of operating works.
An ANN is a computing system composed of multiple layers which are themselves composed of multiple interconnected components called Neurons (or nodes), hence the name neural network. It uses a series of algorithms to try to recognise underlying relationships from a set of data. They have specific parameters defined through training that allows them, when giving them inputs, to perform mathematical operations and determine such relationships.
ANN are used to process information in response to external input (*diagram 1* ***(1)***). This input is processed through a series of layers usually referred to as hidden layers (diagram 1 (2)), the result of which is shown in the output layer (*diagram 1* ***(3)***).

Objective of an ANN

The ANN implements a function which maps inputs to outputs according to an objective. If X is the random variable for the input and Y is the random variable for the output, the ANN models the conditional probability $p_X(Y)$. Usually this objective is to classify inputs (the output is categorical like from an image, the output should say if it is a cat, a dog, a letter A, B, cancer, of healthy cells...), or to perform a regression (the output is continuous, like when we do a regression line between two correlated variables). The parameters of the ANN are

stored in neurons or perceptrons and they are learnt and updated by optimizing the objective when inputs and corresponding outputs are given during the learning stage.

Perceptron & Multilayer Perceptron

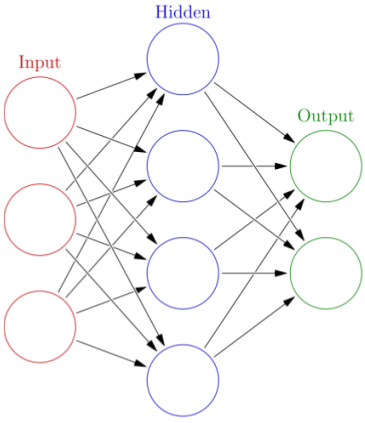
A Perceptron is a single Artificial neuron that takes several inputs (input layer) and produces an output (output layer) (see Diagram 2). It contains two functions: a “Summation function” which is the bias parameter plus the scalar product between the inputs and the weights (or parameters) whose role is to process the inputs given to the perceptron; and an activation function whose role is to add non linearities to the model and to determine which neurons are going to be activated in the following layer (or to get the desired output). Finally a perceptron with 2 inputs is mathematically equivalent to this operations:

$$output = activationfunction(input1 \times weight1 + input2 \times weight2 + bias),$$

Where the *weight1*, *weight2* and bias are the parameters of the neuron. These sommation and activation functions will be discussed more in depth further on.

The structure shown in Diagram 1 is called ‘plain vanilla’ form which is also known as a ‘multilayer perceptron’, it is a network composed of many neurons which are combined and the structure is then called ‘deep’ (see [6], [5], [7]). If we have many inputs and a more consequent amount of data, or when the probabilistic distributions of the input and outputs are very different in this case more neurons are needed to achieve the objective. The ANN in Diagram 1 and 3 are representations of , it shows the structure for when you have multiple Neurons, (interconnected with each other).Keep in mind that although you can only have one input and output layer, it is possible to have more than one Hidden Layer as shown in diagram 3.

Diagram 1 :
A representation of the structure of an Artificial Neural Network [5]



Neural Networks Layers:

- Input layer (1)
- Hidden Layer (2)
- Output Layer (3)

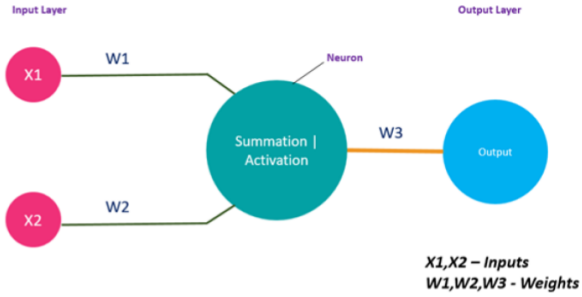


Diagram 2
Representation of a single perceptron [6]

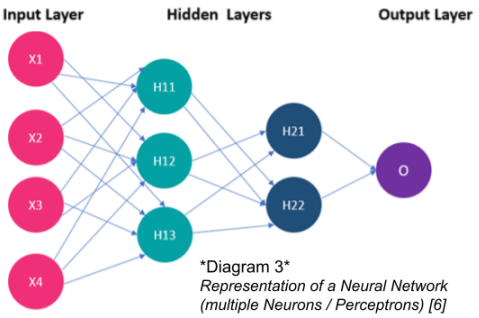


Diagram 3
Representation of a Neural Network (multiple Neurons / Perceptrons) [6]

[7] [3] A scientist named Frank Rosenblatt proposed a simple rule known as ‘Summation’ or ‘weighted sum’ which is followed by an activation function. This rule is used for cases in which there is more than 2 inputs and mathematically described below:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

The threshold value is a real number that is also a parameter of the neuron

How to find the correct parameters: *gradient descent method*

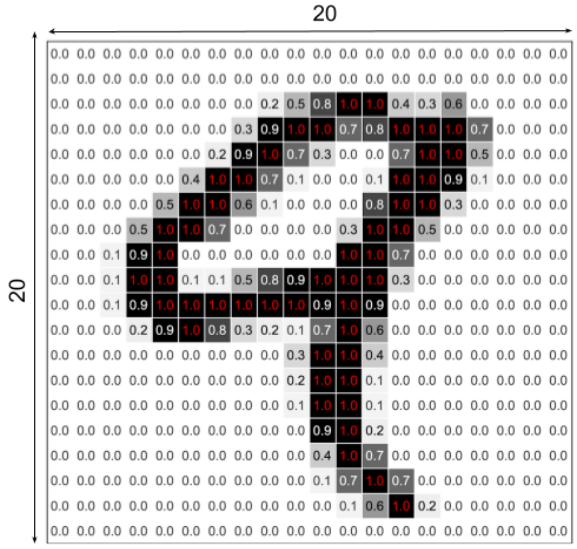
The parameters contained in each neuron are updated and learnt during the “training” stage, when known pairs of inputs and outputs are given. An error function for the ANN is defined to compare the output obtained by the ANN and the desired output. During the training stage the parameters of the network are updated in order to minimize this error function in order to achieve a more accurate result. The goal of the training stage is to reach the minimum amount of errors by updating the weights using the samples of inputs and outputs. This method is called gradient descent, and uses the derivatives of the functions represented by each neuron. The parameters of each neurons are then updated in the direction of the corresponding derivative, with a magnitude defined by the “learning rate” parameter:

$$new\ weight1 = old\ weight1 + learning\ rate \times \frac{d\ errorfunction}{d\ weight1}(error\ obtained)$$

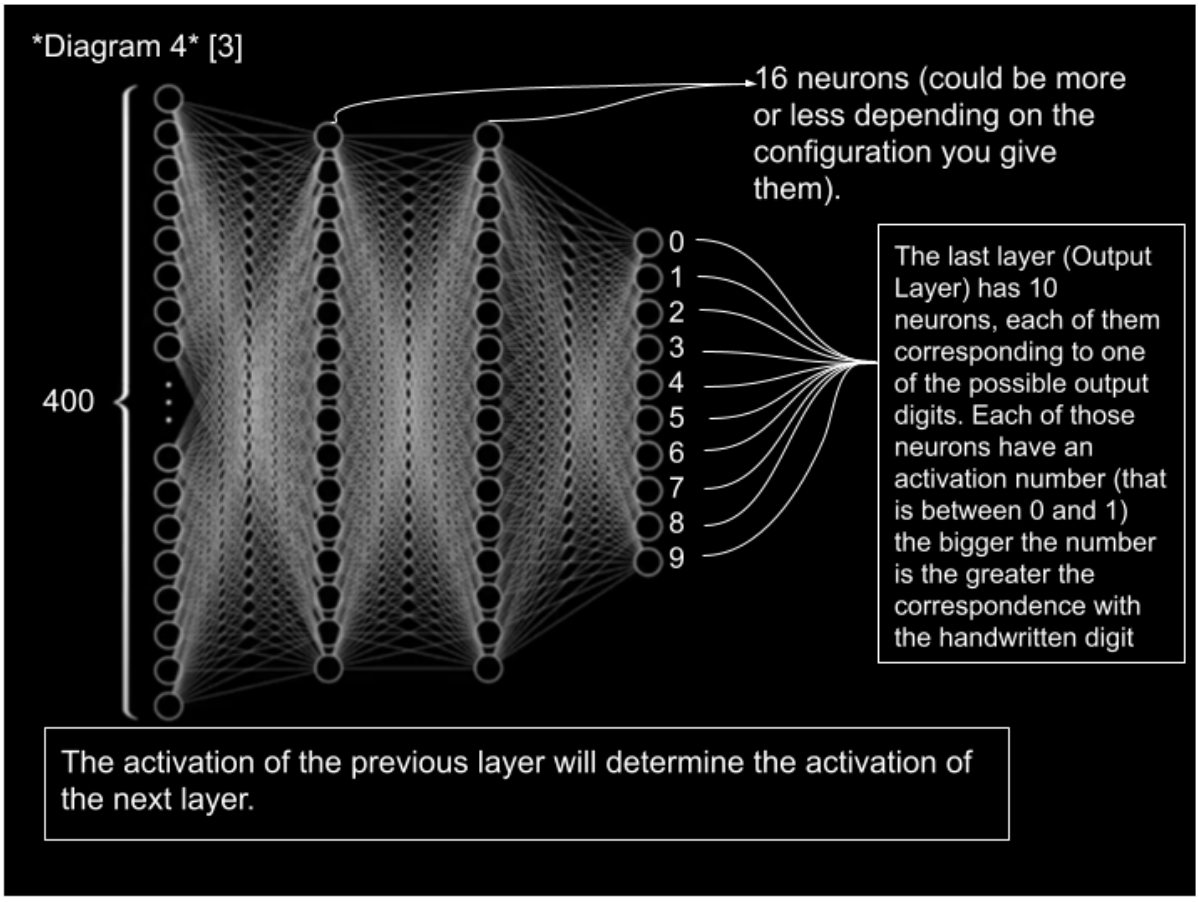
To proceed, the derivatives $\frac{d\ errorfunction}{d\ parameter}$ have to be known. This updating step is called backpropagation of the error in the network by the local derivatives [10].
Then the mathematical computations done in the NN should be differentiable in order to be able to use such a method.

Example and interpretation

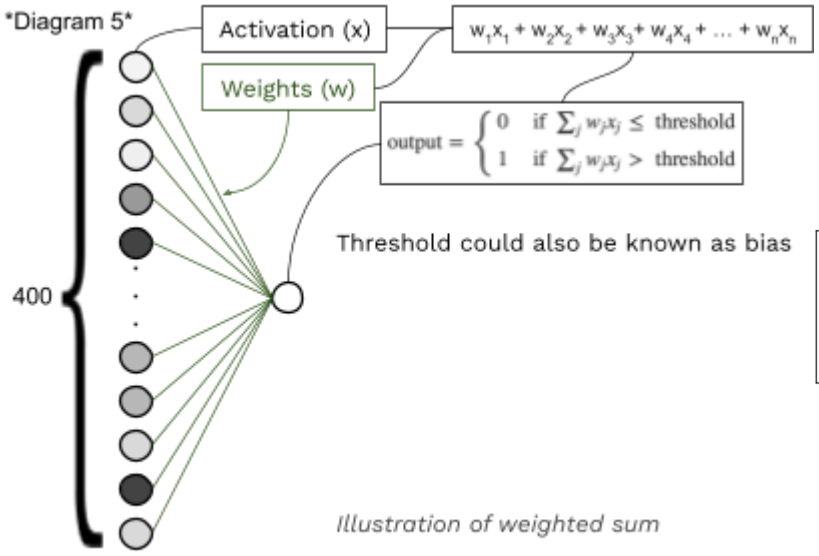
For illustration a neural network could be used in practice to identify handwritten digits. For instance, think about a neuron as a function that outputs a number between 0 and 1 representing levels of grey (0 for white and 1 for black).
Let us consider this image *image 1* (produced it using google sheet representing a handwritten 9) :



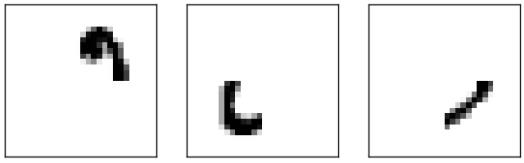
The resolution of this image is 20x20 Pixel, and has $20 \times 20 = 400$ input neurons in totals.The numbers inside the neurons are called their **activation** or their **output** (which is what the following layer of neurons will be inheriting from this layer). Those 400 neurons make up the first layer or input layer of the network (*Diagram 4*).



For this situation, diagram 5 recalls what has been described above with regards to a perceptron with multiple inputs (weighted sum followed by an activation function).



You can assign a greater weight for the most relevant neuron. For example, suppose that at a certain layer we have three neurons corresponding to the following combination of pixels



and that we wish to identify digits such as the ones below
It is clear that for the digit “5”, the third neuron will have a greater weight than the first or the second

one because there is a higher chance and correlation with the third one, whereas for the digits “0” and “9” the second neuron will have a greater weight. We see that an appropriate selection of patterns will allow us to filter all the possibilities until we obtain one definite outcome (the one with the greatest activation in the last layer). The value of the threshold (also known as bias) for a neuron to be activated can be different for each neuron, and this can also be used to make this pattern recognition task more efficient [7][3].

Activation function

The activation function will give a certain value depending on the result of the weighted sum. It must have the following characteristics:

- First of all, it must be a function in the mathematical sense of the word, i.e. it should give a single value as output. Since its input (the result of the weighted sum linking neurons in the previous layer to the neuron in question) is real, what we are looking for is a function mapping real numbers to real numbers: $f : \Re \rightarrow \Re$
- Not only should the outcome of the function should be a real number but a number within some finite activation range, call it $[a,\ b]$
- It should be differentiable in order to be able to use the gradient descent method.

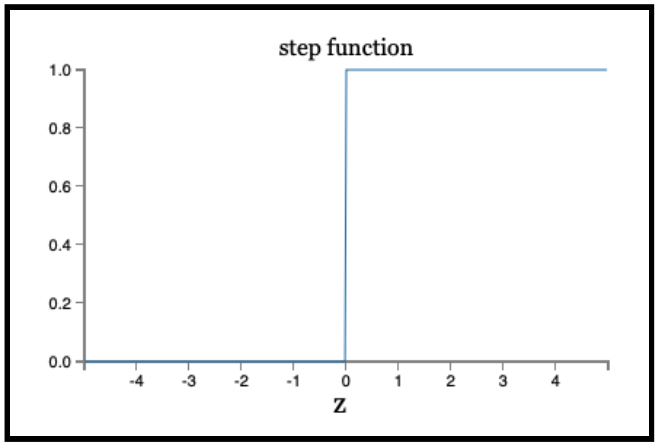
There is multiple ways this can be achieved :

Step function

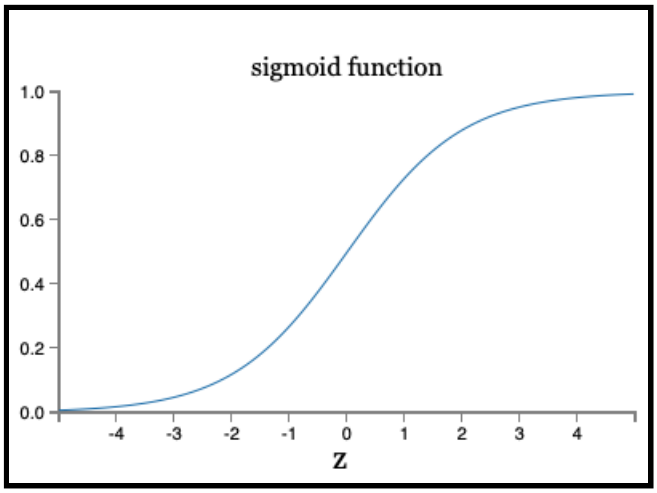
In our example so far, the value of the output was either 0 or 1. Such function might be useful when classifying in two different classes a set of data

$$output = \begin{cases} 0, & z = \sum w_i\, x_i \quad - \text{bias} \leq 0 \\ 1, & z = \sum w_i\, x_i \quad - \text{bias} > 0 \end{cases}$$

Such a behavior could be obtained by means of the step function. If the input (z) is negative then the neuron is not activated (lit), if it’s positive then the neuron is activated. (see graph 1 on the right).



Sigmoid neurons:



In some cases we may be interested in having a less drastic (more gradual) change, so a *derivable* function should be used in order to use the gradient descent method. Ideally, the function should be such that for large negative values it gives approximately zero and for large positive values it gives approximately one.

An example of a function with such a behavior is the sigmoid function (σ), which is a smoothed out version of the step function and differentiable.

The sigmoid function is defined as:

$$\sigma(z) \equiv \frac{1}{1+e^{-z}} \text{ , and } \sigma'(z) \equiv \frac{e^{-z}}{(1+e^{-z})^2}$$

Where $e = 2.718...$ is the Euler number. (see graph 2 on the left)

We can see that for $z \to \infty$ then $e^{-z} \to 0$, $(where \ \to \ = \ tends\ to)$ this means that value of the sigmoid will be $\frac{1}{1+0} = 1$

And inversely for $z \to -\infty$ then $e^{-z} \to \infty$, $(where \ \to \ = \ tends\ to)$ this means that value of the sigmoid will be $\frac{1}{1+\infty} = 0$

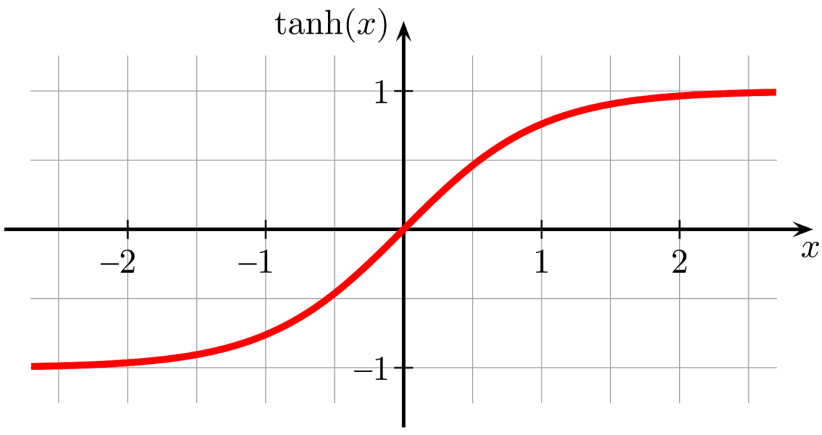
The value could also be between 0 and 1 for example $if\ z = 1$ then $e^{-z} = 1$ this means that the value of the sigmoid will be $\frac{1}{1+1} = 0.5$

Hyperbolic tangent neurons:

If we were still interested in having a less drastic (more gradual) change while the range being [-1, 1]. (Ideally, the function should be such that for large negative values it gives approximately -1 and for large positive values it gives approximately one).

A function that would be appropriate in order to have this behaviour is Tanh which is very similar to the sigmoid function (in fact it is a scaled version) to the exception that tanh is centered at 0 (instead of 0.5) and output range is between -1 and 1 (instead of 0 and 1).

Tanh function is as follows,



$$f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

$$f'(x) = tanh(x) = \frac{4e^{-2x}}{1+e^{-2x}}$$

(see *graph 3* on the right)

For tanh:

If input is a negative number output will be strongly negative.

e.g $(for\ x = -1),\ tanh(x) = -0.761...$

If input is near zero output will be near zero.

e.g $(for\ x = 0),\ tanh(x) = 0$

If input is positive output will be strongly positive.

e.g $(for\ x = 1),\ tanh(x) = 0.761...$

A personal exploration of other potential forms for activations functions:

After studying the previous examples of activation functions I wondered whether there could be other potential forms that would allow to activate neurons in some specific ranges (not necessarily for large values of the input only.)

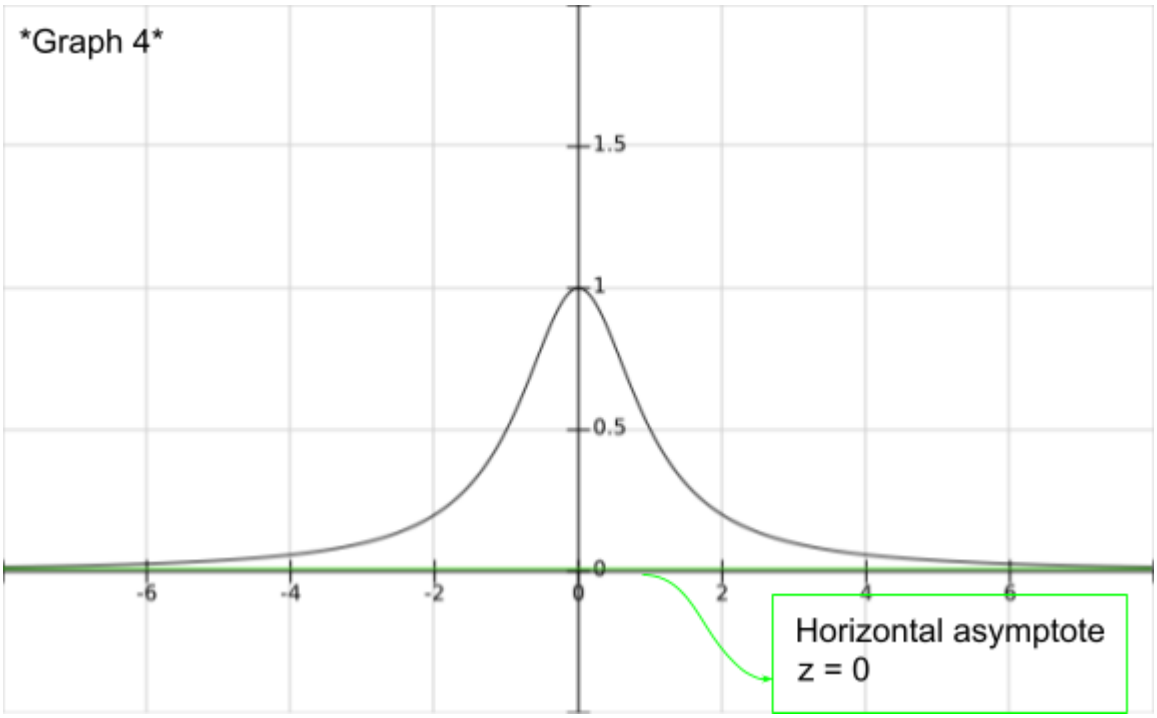
Suppose we wanted a function with the following characteristics:

- Neurons are activated for values near zero and deactivated for both large positive and negative values.
- The magnitude of the activation is between 0 and 1
- Differentiable (for the gradient descent method) and then continuous.

How could we find a function with these characteristics? First of all, if we want the neuron to be deactivated for $z \to \pm\infty$, what we need is a function with a horizontal asymptote at $z = 0$. For that it would suffice to take a function of the type $\frac{P(z)}{Q(z)}$ where $Q(z)$ is a polynomial whose degree is higher than that of $P(z)$ (also a polynomial). A simple example with these characteristics would be $f(z) = \frac{1}{z}$. However, this would not work for two reasons: it gives negative values for $z < 0$, and it goes to infinity when $z = 0$. To solve the first problem we can square z and take instead $\frac{1}{z^2}$, but this does not address the second point. To solve the second point we need to make sure that there are no vertical asymptotes, meaning that $Q(z) \neq 0$ for all z. We can achieve this by adding a constant in the denominator, and take for instance

$$g(z) = \frac{1}{1+z^2} \text{ and } g'(z) = -\frac{2z}{(1+z^2)^2}$$

In Graph 4 we can see the plot of the above function. Large positive and negative values of the input lead indeed to a very small output, and the values near zero are enhanced but limited to a maximum of 1.

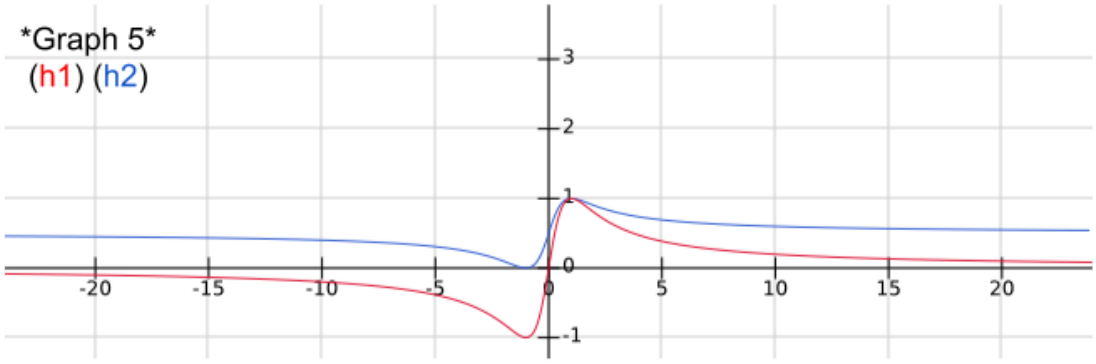


This function satisfies the requirements of our hypothetical situation but does not entirely resemble the examples previously discussed. I noticed, however, that by multiplying $g(z)$ by z I could get a behavior similar to that of the examples. Indeed, if we take

$h_1(z) = \frac{2z}{1+z^2}$ and $h_2(z) = \frac{z}{1+z^2} + \frac{1}{2}$,

And $h_1'(z) = \frac{2(1+z^2)-2z(2z)}{(1+z^2)^2} = \frac{-4z^2+2}{(1+z^2)^2}$ and $h_2'(z) = \frac{1+z^2-z(2z)}{(1+z^2)^2} = \frac{1-z^2}{(1+z^2)^2}$,

we obtain, for values close to $z = 0$, behaviors analogue to that of the hyperbolic tangent and the sigmoid, respectively (see Graph 5) . The analogy breaks for large positive and negative values of z but, precisely because of this reason, the functions would still be interesting for situations where we want enhanced activation in a range close to zero only.



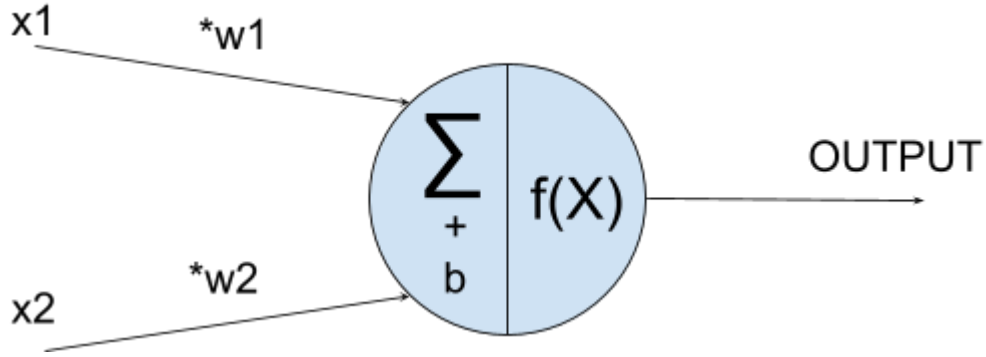
Experimentation with one perceptron and Normal Distribution using python

For this essay an experimentation with one perceptron was carried out. In this experiment we used two normally distributed sets/classes of data generated within the python script (See appendix for the python code). In this case we used cats and dogs as examples for the 2 sets of data. The 2 sets can be considered as 2 dimensional, (a 2D vector (x and y coordinates)), here x and y each represent a different hypothetical characteristics of those animals (for example: Eare size, Weight, Height, Color...).

As we said earlier an ANN needs to be trained with training data in order to calibrate its weights (learn) allowing the ANN to perform the task it was designed for. In this case we try to determine, with a 2D datapoint (x=characteristic1, y=characteristic2) if the point represents a cat or a dog.

Representation and Formula for the perceptron used in the experiment

The perceptron is designed in two parts:
The first part is the scalar product between a 2D input vector (cats or dogs) and a 2 dimensional weight vector. And the second step is the activation function f(X) which is the sigmoid because the objective is to classify.



Mathematical representation of the process that occurs when data goes through the perceptron:

$$\text{When } X = Scalar\,Product + Biais$$

$$X = x_1 w_1 + x_2 w_2 + biais$$

$$\text{And } Output = sigmoid(X)$$

Error function used for the training stage

This is the error function we use for our perceptron:

$$errorfunction(output) = (output - target)^2 = [sigmoid(x_1 w_1 + x_2 w_2 + biais) - target]^2$$

Eq. 12

Where output is the output given by the perceptron, (x1, x2) are the two inputs of the perceptron (x = x1 = characteristic1, y = x2 = characteristic2), and target can be either 0 or 1 depending on what group they are part of (cats or dogs). In the training stage the target is already known before we send in the inputs so that once we get the output of the perceptron we can compare the two and update if needed the decision border so that the model becomes more accurate.

Update rule for the gradient descent method

As said earlier, we need to find the derivatives with respect to the parameters w_1, w_2, $biais$ in order to apply the gradient descent method and update the weights during the training stage, to do so we use the composition rule for derivative:

$$\frac{d\,errorfunction}{d\,w_1} = 2 \times [sigmoid(x_1 w_1 + x_2 w_2 + biais) - target] \times x_1 \times sigmoid'(x_1 w_1 + x_2 w_2 + biais)$$

$$\frac{d\,errorfunction}{d\,w_1} = 2 \times [output - target] \times x_1 \times sigmoid'(X),$$

where $X = x_1 w_1 + x_2 w_2 + biais$ and $output = sigmoid(X)$

So $\frac{d\,errorfunction}{d\,w_1} = 2 \times [output - target] \times x_1 \times \frac{e^{-X}}{(1+e^{-X})^2}$.

In the same way, $\frac{d\,errorfunction}{d\,w_2} = 2 \times [output - target] \times x_2 \times \frac{e^{-X}}{(1+e^{-X})^2}$.

And $\frac{d\,errorfunction}{d\,bias} = 2 \times [output - target] \times \frac{e^{-X}}{(1+e^{-X})^2}$.

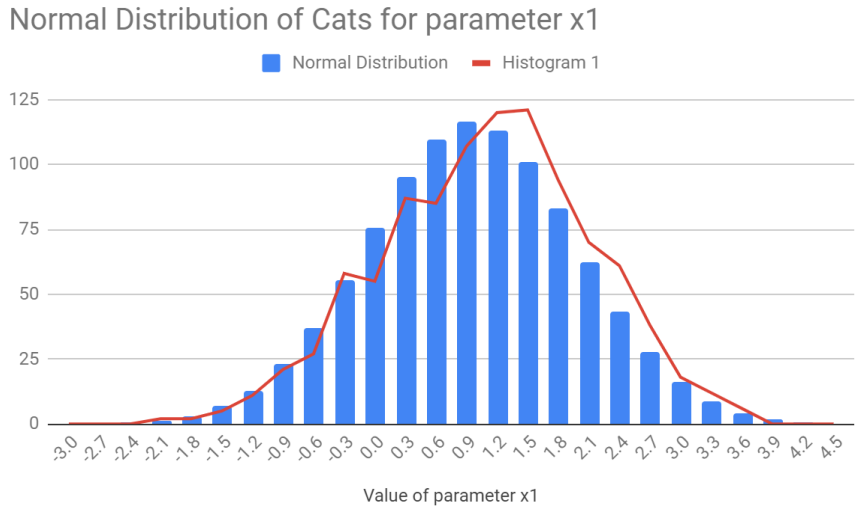
Following the composition rule for the derivative, these derivatives are decomposed in two factors: the derivative of the output error $2 \times [output - target]$ and the local gradients $x_1 \times \frac{e^{-X}}{(1+e^{-X})^2}$, $x_2 \times \frac{e^{-X}}{(1+e^{-X})^2}$ and $\frac{e^{-X}}{(1+e^{-X})^2}$, such that:

$$\frac{d\,errorfunction}{d\,parameter} = (derivative\ of\ the\ output\ error) \times (local\ gradient\ of\ the\ neuron)$$

'' The code I made for this example can be found in the appendix, the sigmoid function is defined in lines 65 to 66, and the derivative of the sigmoid is defined in line 67 to 68. Local gradients are defined in lines 80 to 85. Total derivatives are computed in lines 120 to 121. And finally weights are updated with the function "update" defined in lines 90 to 93. ''

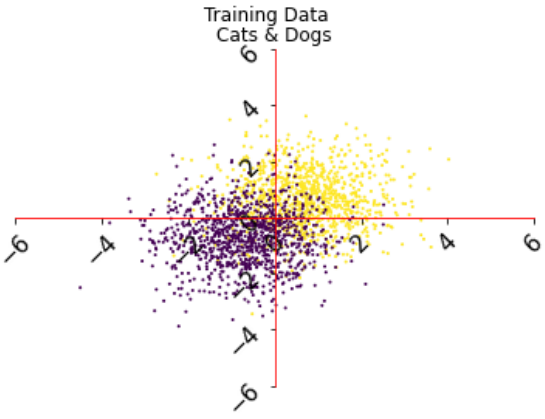
Description of the data simulated

We carried two tests with our ANN, in each we modified slightly the distance between cats and dogs. Both cats and dogs were distributed normally with a standard deviation equal to 1 (spreading) and different means (centered around the mean) (example shown below).

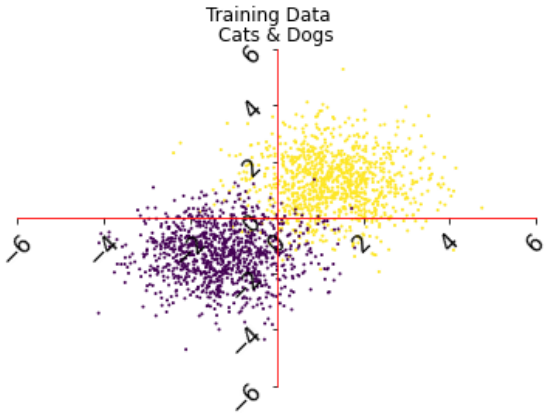


In the first experiment, the two sets of cats and dogs overlap, whereas in the second experiment they are further apart from each other. This is defined in lines 11 & 17 of the code (Cf. appendix, ln 11 & 17), and the graphs below show the repartition of the training data (used to find the parameters) and the testing data (used to find the accuracy of the model) in both cases.

Experiment 1:
mean of cats = 0.7,
mean of dogs = -0.7



Experiment 2:
mean of cats = 1.3,
mean of dogs = -1.3



Results of classification

The accuracy of classification is defined by:

$$accuracy = \frac{number\ of\ correctly\ classified\ inputs}{number\ of\ inputs}$$

Experiment 1

CONFUSION MATRIX		Predicted	
		0	1
Actual	0	88	12
	1	24	76

Test accuracy 82.0%
For instance, there are 24 samples which were incorrectly classified as from the class “0”.

Experiment 2

CONFUSION MATRIX		Predicted	
		0	1
Actual	0	99	1
	1	1	99

Test accuracy 99.0%
For instance, there is 1 sample which is incorrectly classified as from the class “0”.

As expected, the accuracy for experiment 1 gives a higher percentage of correct guesses than experiment 2 as sets are closer in set 2 and more distant in set 1..

Determination of the decision boundary of the Perceptron

This part determines the decision boundary found by the perceptron in order to graphically see how it classifies the data.
The decision border corresponds to an output of 0.5 (y-intercept of the sigmoid function). Let us consider set A and B if, the output of an entry is higher than 0.5 it will be considered as set A and if lower set B. Equation is as follows:

$$Output = sigmoid(x_1 w_1 + x_2 w_2 + biais) = 0.5$$

Below is the sigmoid function part of the activation where e stands for exponential and $X = x_1 w_1 + x_2 w_2 + biais$ which is the result of the scalar product + biais (first part of the neuron).

1. $\frac{1}{1+(e^{-X})} = 0.5$,

In order to find the equation of the vector/decision border, we need to solve Algebraically for *X:*

$$\frac{1}{0.5} = 1 + (e^{-X})$$

$$1 = (e^{-X})$$

We use natural logarithm to find X: $ln(1) = -X$

$$0 = X$$

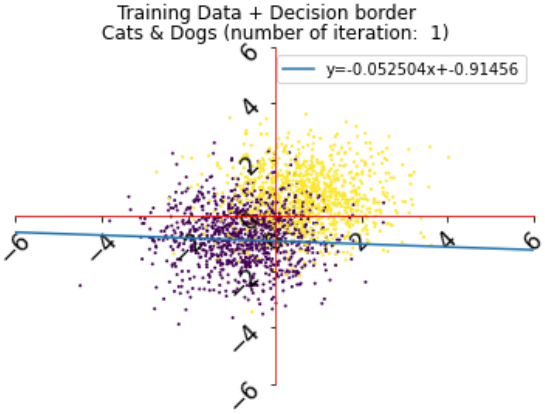
If *X* is equal to 0 then: $0 = x_1 w_1 + x_2 w_2 + biais$

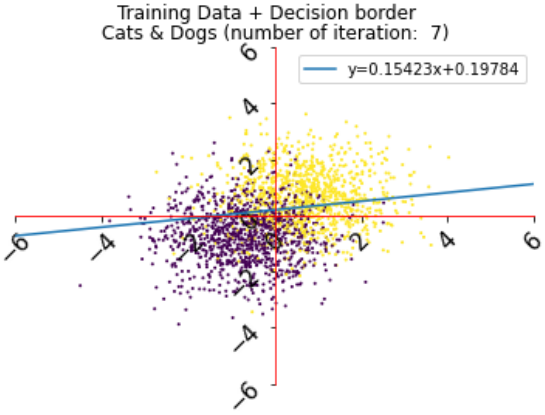
$$-x_1 w_1 - biais = x_2 w_2$$

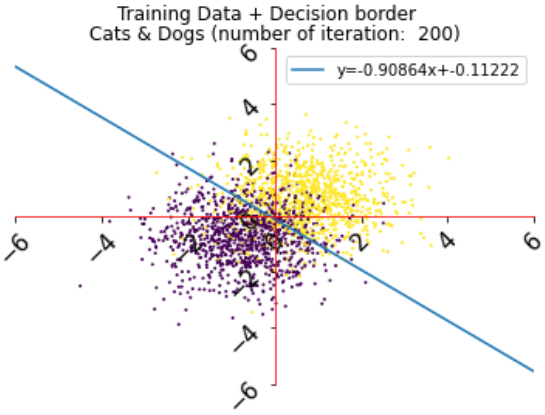
Finally the equation for the decision border can be found in the form ‘y = mx + c’. (w_1, w_2, & biais are fixed values after training). $-\frac{w_1}{w_2} \times x_1 - \frac{b}{w_2} = x_2$ where $y = x_2$, $x = x_1$, $m = -\frac{w_1}{w_2}$ is the gradient of the line, and $c = -\frac{b}{w_2}$

Experiment 1

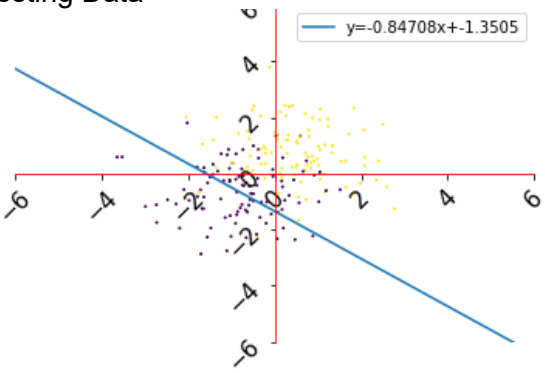
Graphs with decision boundary





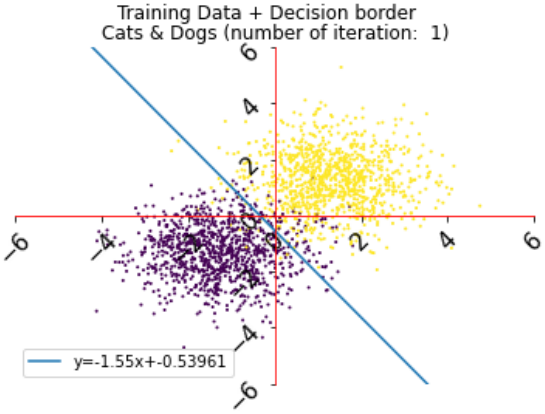


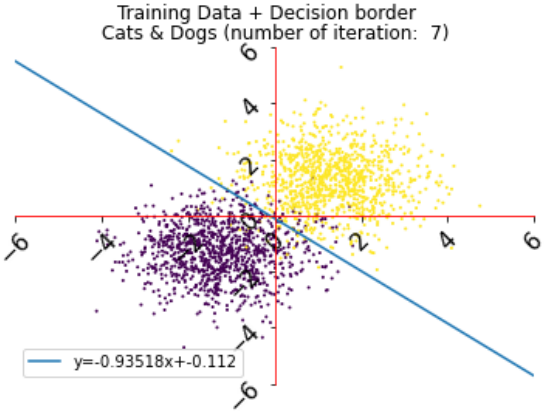
Testing Data

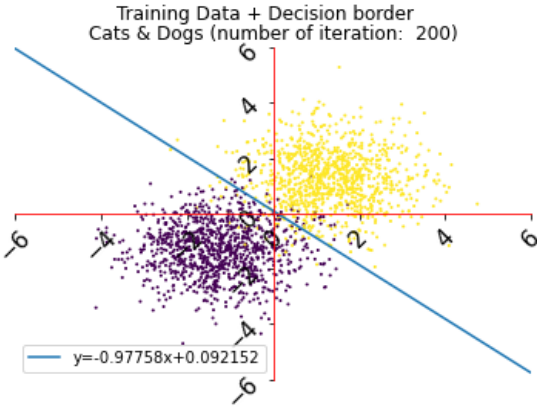


Experiment 2

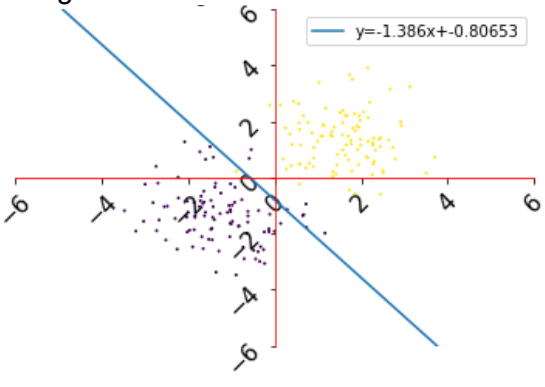
Graphs with decision boundary







Testing Data



This table gives a visualization of what the ANN learns and we can see that the decision boundary between the two classes learns well to separate the two classes of data as it takes at the en the expected optimal position for the classification. We can also see that for the first experiment it is not possible to achieve a high accuracy of classification because the two sets of data overlap.

Evaluation & Conclusion

In this research I explored Artificial Neural Network and the math behind the multiple functions that are in use with regards to our example, namely an ANN that would recognize handwritten digits. I investigated 3 functions which are commonly used in ANN: Sigmoid, Hyperbolic tangent, and Step function. As a result of this exploration I was wondering if there was a way that I could come up with activation functions of my own. In order to find such a function I considered some characteristics, inspired by the behavior of the example functions. For concreteness, I took a hypothetical situation in which input values that are large positive or negative numbers imply that the neuron is deactivated, and inversely small input values that are closer to zero (let's say some number between -1 and 1) the neuron would be activated. I also had to take into account that the magnitude of the activation for my function shall be between 0 and 1. The last but not least, characteristic is that the function needs to be continuous in order to have gradual activation, and even differentiable in order to apply the gradient descent method.

Finding specific applications of an activation function of this type goes beyond the scope of this exploration. Nevertheless, we could imagine situations where we would like to filter patterns that correspond to small (close to zero) values of the weighted sum (for example to identify half-filled regions in a picture).

After performing the calculations and appropriate analysis to propose a possible function ($f(z) = \frac{1}{1+z^2}$) I plotted my function on a graph using the free web-app called fooplot (licensed under GNU, LGPL [9]).

The second part of the work describes and experiments a very simple ANN constituted of a single neuron. We described the mathematical operations performed, including scalar product and the sigmoid activation function, we described how the parameters are learnt from training data and the “gradient descent” method which uses derivatives of each operation performed and we experimented on a simulated classification problem. We showed that a single perceptron is already doing a great job at classifying two different types of 2D data and we investigated how it learns by looking at the decision boundary at each step, until it gets a quite good and expected boundary between the two classes.

The ANN we experimented with has many limitations considering the fact it only possess one neuron/perceptron, which has only two scalar values (2D data, x and y coordinates) as inputs. When the data has more than two inputs such as images which have as many inputs as their number of pixels, or with sounds, the ANN has to be more complex and should integrate more than two inputs, but this is beyond the scope of this research. Moreover, when the data are harder to separate, then the ANN sould be composed of more neurons/perceptrons, and hence have a greater complexity and more parameters to learn, although this is also outside the scope of this essay. When more neurons are added in the ANN, it will be much harder to find and visualize the equation of the border of decision,which won’t be a line anymore but more like a curve; as a possible future extension of this essay it would be very interesting to investigate such a scenario. More complex neurons also make use of matrix multiplications instead of scalar products which is much more complicated than what we have seen so far, and therefore falls once again beyond the scope of this essay as we only investigated neurons with scalar products. Machine Learning integrates different mathematical aspects, in the literature there are often references to probabilities and many other unseen mathematical domains that can explain much more complex models than the

ones that we have seen in this essay. In the future it would be interesting to keep learning more about this subject and improving the basic model investigated in this research.

REFERENCES

[1] : Wikipedia contributors, 'Neural network', *Wikipedia, The Free Encyclopedia,* 30 November 2020, 17:54 UTC,<https://en.wikipedia.org/w/index.php?title=Neural_network&oldid=991555932> [accessed 2 December 2020]

[2] : Chen, J 2020, *Neural Network*, viewed 2 December 2020, <https://www.investopedia.com/terms/n/neuralnetwork.asp>.

[3] : 3Blue1Brown 2017, *But what is a Neural Network?* | *Deep learning, chapter 1*, online video, 5 October, viewed 2 December 2020, <https://youtu.be/aircAruvnKk>.

[4] : Euge Inzogarat, 2018, Understanding Neural Networks: What, How and Why?, Towards Data science, 30 October, viewed 2 December 2020 <https://towardsdatascience.com/understanding-neural-networks-what-how-and-why-18ec703ebd31>

[5] : By Glosser.ca - Own work, Derivative of File:Artificial neural network.svg, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=24913461>

[6] : Sampath kumar gajawada 2019, 'The Math behind Artificial Neural Networks', Towards Data science, 17 November, viewed 3 December 2020, <https://towardsdatascience.com/the-heart-of-artificial-neural-networks-26627e8c03ba>.

[7] : Neural Networks and Deep Learning", Determination Press, 2015,<http://neuralnetworksanddeeplearning.com/chap1.html>, viewed 4 December 2020

[8] : By Geek3 — Personal work, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=4198479>

[9] : FooPlot <http://fooplot.com>, viewed 5 December 2020

[10] : Gradient descent and backpropagation,Tobias Hill, Dec 4, 2018, https://towardsdatascience.com/part-2-gradient-descent-and-backpropagation-bf90932c066a

APPENDIX

The ANN Python script I made for this IA:

The comments & explanation are the part that are highlighted in purple: # example

```
1. import numpy as np
2. import pandas as pd
3. import os
4. from matplotlib import pyplot as plt  # convention on importing
   pyplot
5. from tqdm import tqdm
6. # We assume that cats are 2D data points centered around the
   point of coordinate (2,2)and are normally distributed and dogs
   are also normally distributed but around the points (-2,-2)
7. # The reason for both cats and dogs being 2D data points is
   because it corresponds to 2 features describing these animals,
   such features could be for example Eyes Spacing, Weights,
   Heights, Ears, etc...
8. # Let us consider the previous described data model (points)
9. # Simulating/Generating TRAINING data (Used to find the value of
   the weights (Cats)
10.    n_train = 1000
11.    training_cats = np.random.randn(n_train, 2) + 2 * np.ones((1,
   1))  # pos
12.    # Pos stands for positive
13.    # print(training_cats[:10]) # Print statement for testing
   purpose
14.    # print(training_cats.shape) # Print statement for testing
   purpose
15.    # print(np.mean(training_cats, 0)) # Print statement for
   testing purpose
16.    # Simulating/Generating TRAINING data (Used to find the value
   of the weights (Dogs)
17.    training_dogs = np.random.randn(n_train, 2) - 2 * np.ones((1,
   1))  # neg
18.    # neg stands for negative
19.    training_cats = np.concatenate([training_cats, [[1]] *
   n_train], 1)
20.    training_dogs = np.concatenate([training_dogs, [[0]] *
   n_train], 1)
21.    # Concatenating the training data
22.    training_data = np.concatenate([training_cats, training_dogs],
   0)
23.    plt.scatter(training_data[:, 0], training_data[:, 1],
   c=training_data[:, 2], s=1)
24.    plt.title('Cats & Dogs')
25.    plt.suptitle('Training Data')
26.    ax = plt.gca()
27.    ax.spines['right'].set_color('none')
28.    ax.spines['top'].set_color('none')
```

```
29.    ax.xaxis.set_ticks_position('bottom')
30.    ax.spines['bottom'].set_position(('data', 0))
31.    ax.spines['bottom'].set_color(('red'))
32.    ax.yaxis.set_ticks_position('left')
33.    ax.spines['left'].set_position(('data', 0))
34.    ax.spines['left'].set_color(('red'))
35.    plt.xticks(fontsize=15, rotation=45)
36.    plt.yticks(fontsize=15, rotation=45)
37.    plt.show()
38.    # (.randn is to make normal distribution)
39.    # (n_train is the number of samples)
40.    # 2 is the size of the sample (2 parameters / 2D data points))
41.    # Simulating/Generating TESTING data (Used to evaluate the
    value of the weights/model (Cats))
42.    n_test = 100
43.    testing_cats = np.random.randn(n_test, 2) + 2 * np.ones((1, 1))
    # pos
44.    # Simulating/Generating TESTING data (Used to evaluate the
    value of the weights/model (Cats))
45.    testing_dogs = np.random.randn(n_test, 2) - 2 * np.ones((1, 1))
    # neg
46.    testing_cats = np.concatenate([testing_cats, [[1]] * n_test],
    1)
47.    testing_dogs = np.concatenate([testing_dogs, [[0]] * n_test],
    1)
48.    testing_data = np.concatenate([testing_cats, testing_dogs], 0)
49.    np.random.shuffle(testing_data)
50.    plt.scatter(testing_data[:, 0], testing_data[:, 1],
    c=testing_data[:, 2], s=1)
51.    ax = plt.gca()
52.    plt.title('Cats & Dogs')
53.    plt.suptitle('Testing Data')
54.    ax.spines['right'].set_color('none')
55.    ax.spines['top'].set_color('none')
56.    ax.xaxis.set_ticks_position('bottom')
57.    ax.spines['bottom'].set_position(('data', 0))
58.    ax.spines['bottom'].set_color(('red'))
59.    ax.yaxis.set_ticks_position('left')
60.    ax.spines['left'].set_position(('data', 0))
61.    ax.spines['left'].set_color(('red'))
62.    plt.xticks(fontsize=15, rotation=45)
63.    plt.yticks(fontsize=15, rotation=45)
64.    plt.show()
65.    def sigmoid(x):
66.        return 1 / (1 + np.exp(-x))
67.    def derivative_sigmoid(x):
68.        return np.exp(-x) / (1 + np.exp(-x)) ** 2
69.    class Perceptron():
70.        def __init__(self, weights, bias):
71.            self.weights = np.asarray(weights)
72.            self.bias = bias
73.        def forward(self, x1, y2):
74.            middleP = x1 * self.weights[0] + x2 * self.weights[1] +
    self.bias  # scalar product + bias
75.            outputP = sigmoid(middleP)  # Activation function
76.            if outputP > 0.5:  # Classification function
77.                outputC = 1
78.            else:
```

```
            outputC = 0
        # Local gradients
        lg = np.asarray([
            x1 * derivative_sigmoid(middleP),
            x2 * derivative_sigmoid(middleP),
            derivative_sigmoid(middleP)
        ])
        return outputP, outputC, lg
    def backprop(self, derr_out, lg):
        # backpropagates the gradients
        return derr_out * lg
    def update(self, descent):
        # update the weights by the gradient descent method
        self.weights -= descent[:2]
        self.bias -= descent[-1]
    def get_weights(self):
        return self.weights, self.bias
# ------------------- ANN DEFINITION ----------------------------------------
lr = 1  # learning rate
bias = np.random.rand()  # Value bias
weights = [np.random.rand(), np.random.rand()]  # Weights
oneP_ann = Perceptron(weights, bias)
# ------------------- TRAINING + EPOCH EVALUATION ----------------------------
epochs = 1  # number of epochs
batch = 1  # size of the batches
# number of batches and training data reduction
n_batches = int(training_data.shape[0] // batch)
for epoch in range(epochs):
    # shuffling, and batching the training data
    np.random.shuffle(training_data)
    etraining_data = training_data[:batch * n_batches]  # reduction
    etraining_data = etraining_data.reshape(n_batches, batch, 3) # batching
    pbar = tqdm(total=len(etraining_data), ascii=True, desc="Epoch {} / Error {}".format(epoch + 1, np.nan))
    for training_batch in etraining_data:
        errors = []
        gradients = []
        for sample in training_batch:
            # sample[:2] is the input and sample[-1] is the objective for output
            outputP, _, lg = oneP_ann.forward(*sample[:2])  # apply the ann on the input
            error = (outputP - sample[-1]) ** 2  # compute the error
            errors.append(error)
            derror = 2 * (outputP - sample[-1])  # compute the derivative of the error
            gradient = oneP_ann.backprop(derror, lg)  # backpropagate the derivative to the weights
            gradients.append(gradient)
        gradients = np.array(gradients)
        batch_error = np.mean(errors)
        batch_gradients = np.mean(gradients, 0)
        # print(batch_gradients)
```

```
127.             # update the weights
128.             oneP_ann.update(batch_gradients)
129.             # print(oneP_ann.get_weights())
130.             pbar.set_description_str(desc="Epoch {} / Error
    {}".format(epoch + 1, batch_error))
131.             pbar.update()
132.         pbar.close()
133.         testing_goals = []
134.         testing_results = []
135.         for sample in testing_data:
136.             testing_goals.append(sample[-1])
137.             _, outputC, _ = oneP_ann.forward(*sample[:2])
138.             testing_results.append(outputC)
139.         testing_goals = np.asarray(testing_goals)
140.         testing_results = np.asarray(testing_results)
141.         confusion_matrix = pd.crosstab(
142.             testing_goals,
143.             testing_results,
144.             rownames=['Actual'],
145.             colnames=['Predicted'])
146.         print("\n")
147.         print(confusion_matrix)
148.         print("Epoch {} Test acccuracy {}%\n\n".format(epoch,
    np.sum(testing_goals == testing_results) / len(testing_goals) *
    100))
```