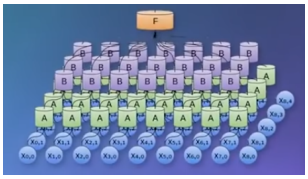
Introduction to Neural Networks

Table of Contents

- 3.1 Introduction
- 3.2 Classification Problems 1
- 3.3 Classification Problems 2
- 3.4 Linear Boundaries
- 3.5 Higher Dimensions
- 3.6 Perceptrons
- 3.7 Why Neural Networks?
- 3.8 Perceptrons as Logical Operators
- 3.9 Perceptron Trick
- 3.10 Perceptron Algorithm
- 3.11 Non Linear Regions
- 3.12 Error Functions
- 3.13 Log-loss Error Function
- 3.14 Discrete vs Continous

3.1 Introduction

- **What is Deep Learning?, What is it used for?**
    - **Deep Learning is subfield of machine learning concerned with algorithms inspired by the structure and function of the brain called Artificail Neural Networks.**
    - **It has many applications such as:**
        - beating Humans in games such as Go or jeopardy
        - Detecting spam in emails
        - forcasting stock prices
        - recognizig images in pictures
        - diagnosing illnesses sometimes with more percision than doctors
        - self-driving cars
- **Neural Networks**
    - It vaguely mimic the process of how the brain operates, with neurons that fire bits of information.



3.2 Classification Problems 1

When we have a system for acceptance of students at a university and most people who get 9 in test and 8 in grades more likely to get accepted but who get 3 in test and 4 in grades are more likely not to be accepted, so If we have a student who gets 7 in test and 6 in grades, will he be accepted or not?

3.3 Classification Problems 2

we could say from the data above that the student will be accepted because he falls at the area of accepted students. We can clarify it more by setting a line which seperates the students who got accepted or not accepted like below

3.4 Linear Boundaries

that line which has been drawn it has an equation $2x_1 + x_2 - 18 = 0$ which mean to accept or reject a student we should see the equation result $2 * test + grades - 18 = 0$ and the result if it > 0 the student will be accepted and if it < 0 the student will be rejected

BOUNDARY:
A LINE
$2x_1 + x_2 - 18 = 0$

Score =
2*Test + Grades - 18

PREDICTION:
Score > 0: Accept
Score < 0: Reject

In more general case the equation of the boundary line will be $w_1x_1 + w_2x_2 + b = 0$ and to summarize it more, we will have $WX + b = 0$ which W is a vector of w_1, w_2 and X is a vector of x_1, x_2 and y is a label of 0 or 1. The prediction variable $\hat{y}$ which will be 1 if $WX + b \geq 0$ which will be above the line and will be 0 if $WX + b < 0$ which will be below the line.

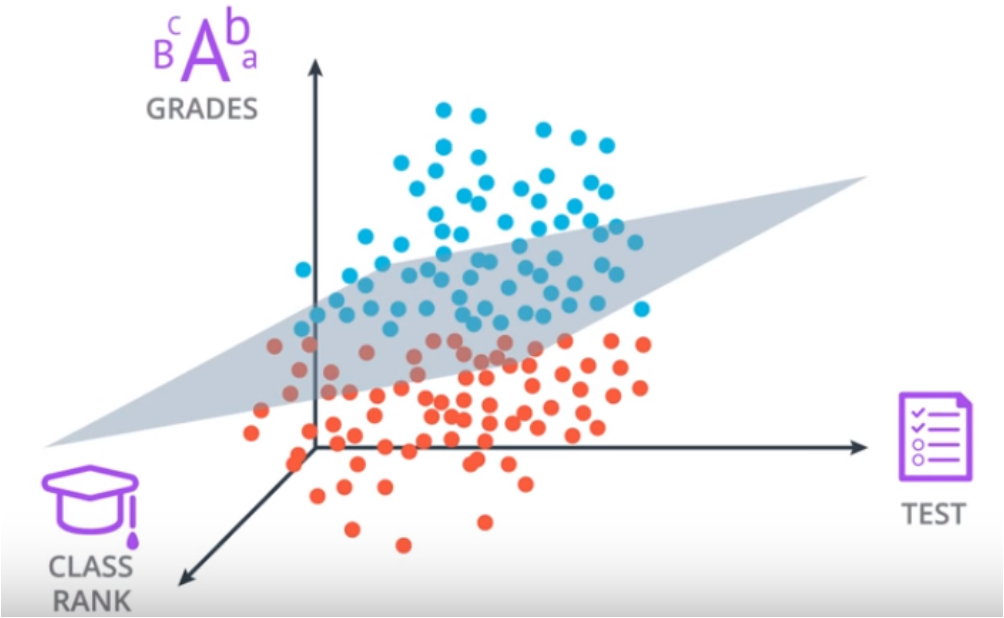
BOUNDARY:
A LINE
$w_1x_1 + w_2x_2 + b = 0$
$Wx + b = 0$
$W = (w_1, w_2)$
$x = (x_1, x_2)$
y = label: 0 or 1

PREDICTION:

$$\hat{y} = \begin{cases} 1 \text{ if } Wx + b \geq 0 \\ 0 \text{ if } Wx + b < 0 \end{cases}$$

3.5 Higher Dimensions

If we have 3 columns instead of 2, we won't be working in 2 dimensions, we will be working in three dimensions



The Equation for that plane will be $w_1x_1 + w_2x_2 + w_3x_3 + b = 0$ but it still could be abreviated with $WX + b = 0$ but instead the vector $\vec{W}$ will include w_1, w_2, w_3 and the vector $\vec{X}$ will include x_1, x_2, x_3 and the predction will still

$$\hat{y} \begin{cases} 1 \text{ if } WX + b \geq 0 \\ 0 \text{ if } WX + b < 0 \end{cases}$$

But what if we have n dimensional space $x_1, x_2, \ldots, x_n$, We will have n dimensional hyperplane and the equation will be $w_1x_1 + w_2x_2 + w_nx_n + b = 0$ but it still could be abreviated with $WX + b = 0$ but instead the vector $\vec{W}$ will include $w_1, w_2, \ldots, w_n$ and the vector $\vec{X}$ will include $x_1, x_2, \ldots, x_n$ and the predction will still

$$\hat{y} \begin{cases} 1 \text{ if } WX + b \geq 0 \\ 0 \text{ if } WX + b < 0 \end{cases}$$

n-dimensional space
$x_1, x_2, \ldots, x_n$

BOUNDARY:
n-1 dimensional hyperplane
$w_1x_1 + w_2x_2 + w_nx_n + b = 0$
$Wx + b = 0$

PREDICTION:

$\hat{y} = \begin{cases} 1 \text{ if } Wx + b \geq 0 \\ 0 \text{ if } Wx + b < 0 \end{cases}$

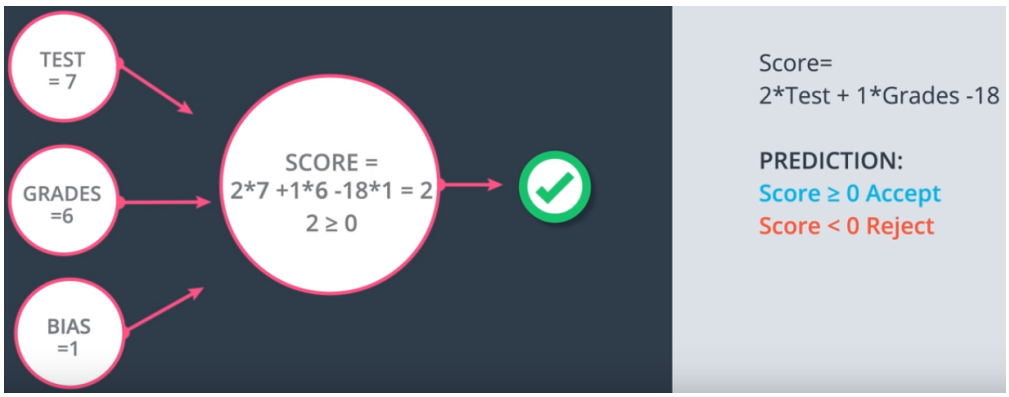
BOUNDARY:
A PLANE
$w_1x_1 + w_2x_2 + w_3x_3 + b = 0$
$Wx + b = 0$

PREDICTION:

$\hat{y} = \begin{cases} 1 \text{ if } Wx + b \geq 0 \\ 0 \text{ if } Wx + b < 0 \end{cases}$

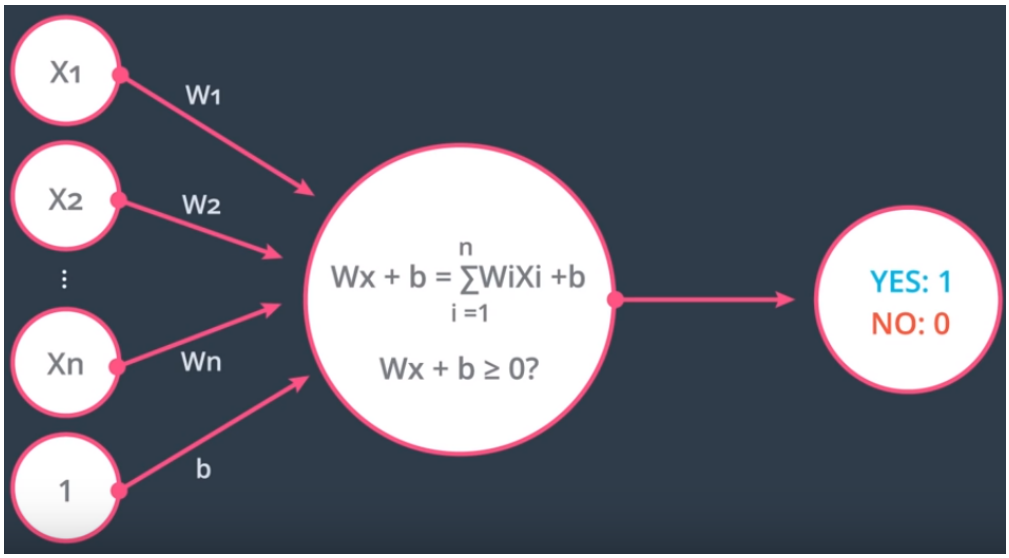
3.6 Perceptrons

It's a neural network unit which make some computations on the data to extract features from it



As we see in the above graph, we input the data to the perceptron unit to evaluate the inputs and classify if it belongs to accepted or rejected area and we use the score fuction $2 * Test + 1 * Grades - 18$ to determine that if $Score \geq 0$ Accepted and if $Score < 0$ Rejected.

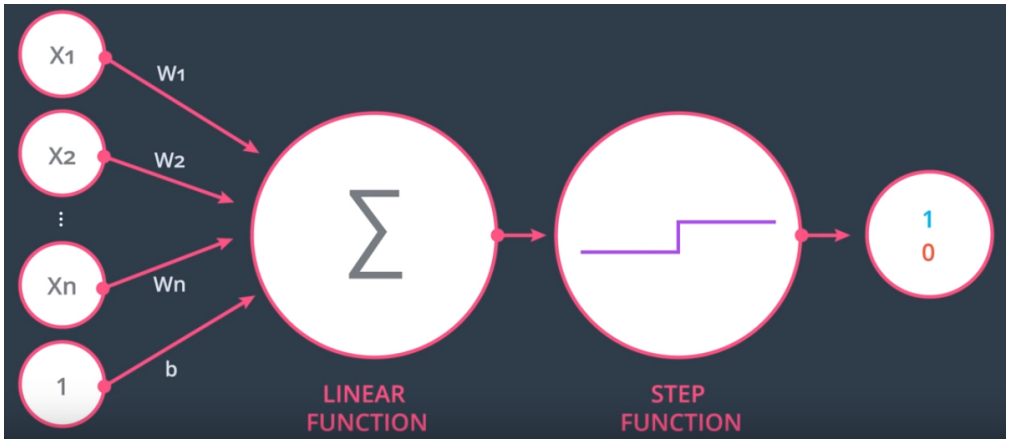
But in the General Case the node will have an input values $x_1, x_2, \ldots, x_n$ and 1 and edges with wieghts $w_1, w_2, \ldots, w_n$ and b corresponding for bais unit then the node will calculate the linear euqation $WX + b = \sum_{i=1}^{n} W_iX_i + b$ then it checks if $WX + b \geq 0$ if it is, then the node returns a value of one for yes and if not then it returns a value of zero for no.



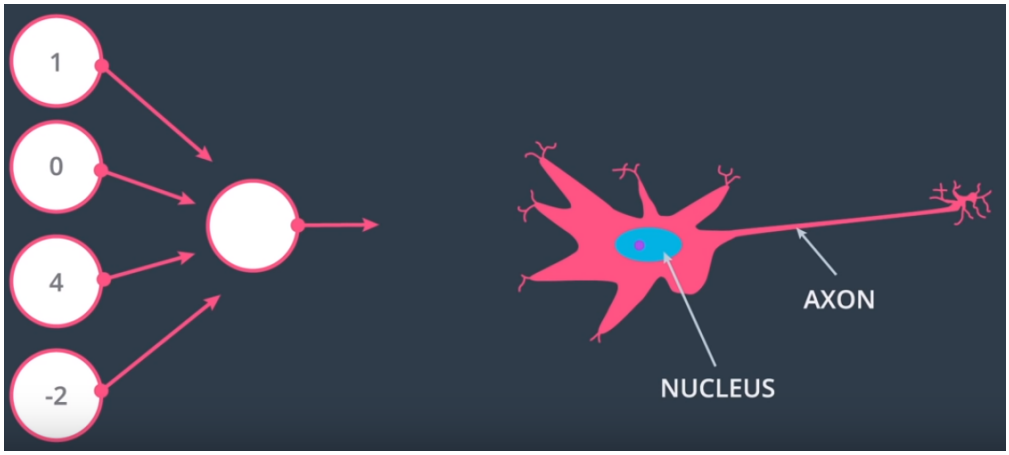
The step of evaluation whether it 's 0 or 1, It's called "Step Function" which returns

$$y = \begin{cases} 1 \text{ if } x \geq 0 \\ 0 \text{ if } x < 0 \end{cases}$$

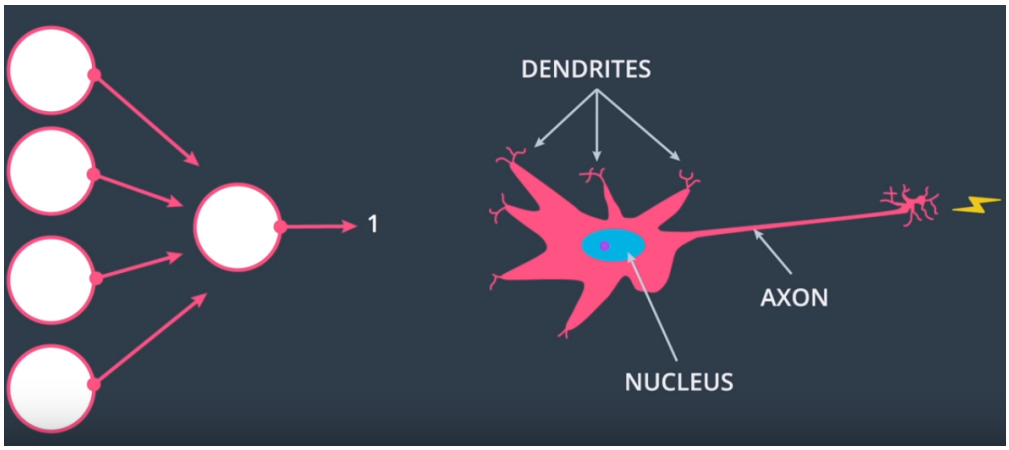
, so we could say that those perceptrons are a combination of nodes the first one as a "Linear Function" and the second on is a "Step Function"



3.7 Why Neural Networks?



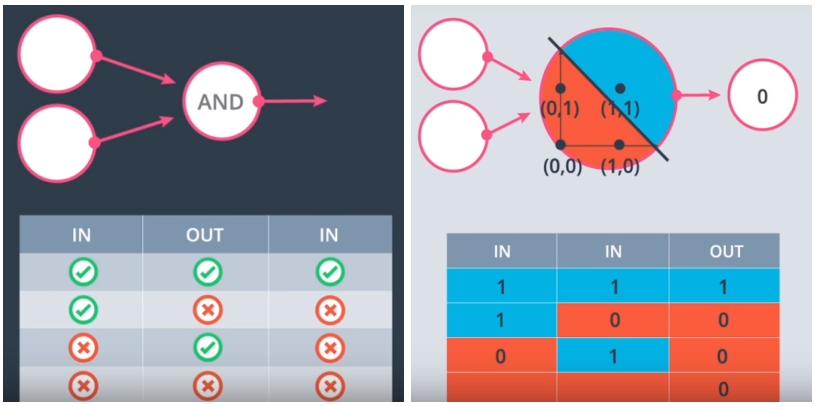
The reason why these objects called neural networks because perceptions kind of look like neurons in the brain.



The percptron get the data inputs and does some calculations on them to output 0 ot 1, just as neurons get the impulses through dendrites and do something to it and then decides if it outputs a nervous impulse or not through axon

3.8 Perceptrons as Logical Operators

- AND Perceptron
    - **It takes 2 inputs and evaluates to true when the 2 inputs are True and To apply these into perceptron we draw a table of zeros and ones where negative area corresponds to red and positive to blue and it evaluates to 1 when the two inputs are 1**

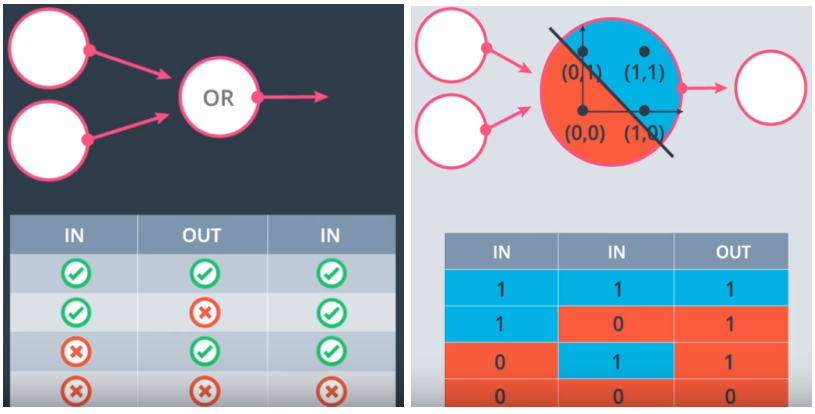



```
import pandas as pd

# TODO: Set weight1, weight2, and bias
weight1 = 1.0
weight2 = 1.0
bias = -2.0


# DON'T CHANGE ANYTHING BELOW
# Inputs and outputs
test_inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]
correct_outputs = [False, False, False, True]
outputs = []

# Generate and check output
for test_input, correct_output in zip(test_inputs, correct_outputs):
```

```
    linear_combination = weight1 * test_input[0] + weight2 * test_input[1] + bias
    output = int(linear_combination >= 0)
    is_correct_string = 'Yes' if output == correct_output else 'No'
    outputs.append([test_input[0], test_input[1], linear_combination, output,
is_correct_string])

# Print output
num_wrong = len([output[4] for output in outputs if output[4] == 'No'])
output_frame = pd.DataFrame(outputs, columns=['Input 1', '  Input 2', '  Linear
Combination', '  Activation Output', '  Is Correct'])
if not num_wrong:
    print('Nice!  You got it all correct.\n')
else:
    print('You got {} wrong.  Keep trying!\n'.format(num_wrong))
print(output_frame.to_string(index=False))
```

```
Nice!  You got it all correct.

Input 1    Input 2    Linear Combination    Activation Output    Is Correct
      0          0                  -2.0                    0           Yes
      0          1                  -1.0                    0           Yes
      1          0                  -1.0                    0           Yes
      1          1                   0.0                    1           Yes
```

- OR Perceptron
    - **It takes 2 inputs and evaluates to True when one of the inputs is True and To apply these into perceptron we draw a table of zeros and ones where negative area corresponds to red and positive to blue and it evaluates to 1 when the one of inputs are 1**




```
import pandas as pd

# TODO: Set weight1, weight2, and bias
weight1 = 2.0
weight2 = 2.0
bias = -1.0
```

```
# DON'T CHANGE ANYTHING BELOW
# Inputs and outputs
test_inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]
correct_outputs = [False, False, False, True]
outputs = []

# Generate and check output
for test_input, correct_output in zip(test_inputs, correct_outputs):
    linear_combination = weight1 * test_input[0] + weight2 * test_input[1] + bias
    output = int(linear_combination >= 0)
    is_correct_string = 'Yes' if output == correct_output else 'No'
    outputs.append([test_input[0], test_input[1], linear_combination, output,
is_correct_string])

# Print output
num_wrong = len([output[4] for output in outputs if output[4] == 'No'])
output_frame = pd.DataFrame(outputs, columns=['Input 1', '  Input 2', '  Linear
Combination', '  Activation Output', '  Is Correct'])
if not num_wrong:
    print('Nice!  You got it all correct.\n')
else:
    print('You got {} wrong.  Keep trying!\n'.format(num_wrong))
print(output_frame.to_string(index=False))
```

```
Nice!  You got it all correct.

Input 1    Input 2    Linear Combination    Activation Output   Is Correct
      0          0                  -1.0                    0          Yes
      0          1                   1.0                    1          Yes
      1          0                   1.0                    1          Yes
      1          1                   3.0                    1          Yes
```

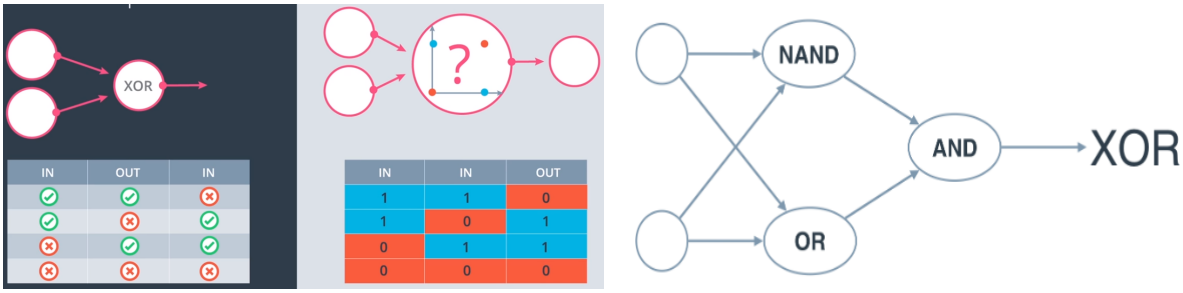
- NOT Perceptron
    - **Unlike other operators we looked at, the NOT operation only cares about one input. the operation returns a 0 if the input is 1 and a 1 if the input is 0. The other inputs to perceptron are ignored**

```
import pandas as pd

# TODO: Set weight1, weight2, and bias
weight1 = 1.0
weight2 = -2.0
bias = 0.0


# DON'T CHANGE ANYTHING BELOW
# Inputs and outputs
```

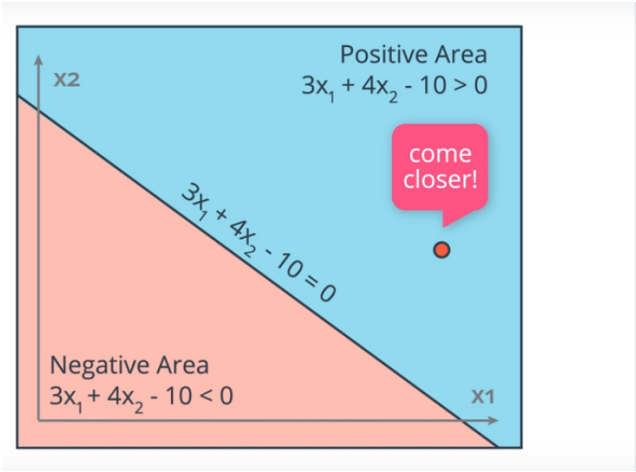
```
test_inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]
correct_outputs = [True, False, True, False]
outputs = []

# Generate and check output
for test_input, correct_output in zip(test_inputs, correct_outputs):
    linear_combination = weight1 * test_input[0] + weight2 * test_input[1] + bias
    output = int(linear_combination >= 0)
    is_correct_string = 'Yes' if output == correct_output else 'No'
    outputs.append([test_input[0], test_input[1], linear_combination, output,
is_correct_string])

# Print output
num_wrong = len([output[4] for output in outputs if output[4] == 'No'])
output_frame = pd.DataFrame(outputs, columns=['Input 1', '  Input 2', '  Linear
Combination', '  Activation Output', '  Is Correct'])
if not num_wrong:
    print('Nice!  You got it all correct.\n')
else:
    print('You got {} wrong.  Keep trying!\n'.format(num_wrong))
print(output_frame.to_string(index=False))
```

```
Nice!  You got it all correct.

Input 1    Input 2    Linear Combination    Activation Output    Is Correct
      0          0                   0.0                    1           Yes
      0          1                  -2.0                    0           Yes
      1          0                   1.0                    1           Yes
      1          1                  -1.0                    0           Yes
```

- XOR Perceptron
    - **It's a multilayer perceptron which takes 2 inputs to the first layer which consists of AND, NOT and OR then feed the results to AND perceptron to get XOR perceptron.**



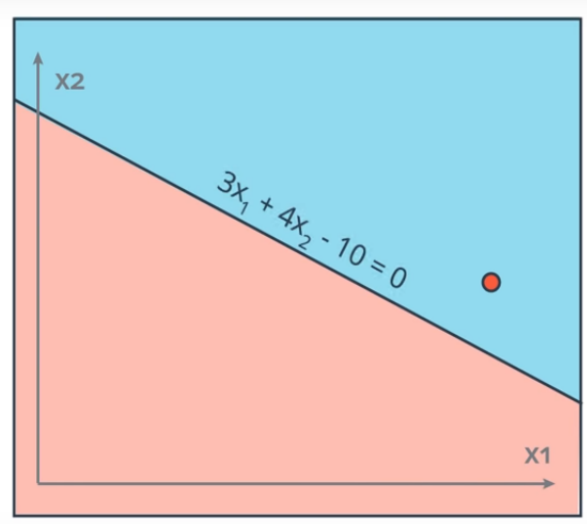
3.9 Perceptron Trick



LINE: $3x_1 + 4x_2 - 10 = 0$
POINT: (4,5)

	3	4	-10
−	4	5	1
	-1	-1	-11

If we have a line with function $2.6x_1 + 4x_2 - 10 = 0$ and a point in coordinates $(4, 5)$ and the point in the positive area which is misclassified and should be in the negative area, so we need to make the line near from the point without misclassified the other points, if $\begin{matrix} 3 & 4 & -10 \\ - & & \\ 4 & 5 & 1 \end{matrix}$ we will have $-1 \quad -1 \quad -11$ which will make the line shift aggressively and we could misclassify the other points, so we could use what's called learning rate (0.1) which will help the line move short steps towards the point $\begin{matrix} 3 & 4 & -10 \\ - & & \\ 4*0.1 & 5*0.1 & 1*0.1 \end{matrix}$ and we will get a new line $2.6x_1 + 3.5x_2 - 10.1 = 0$ and we do the same steps if the point was in the negative area but we will add the points not subtract them.



LINE: $3x_1 + 4x_2 - 10 = 0$
POINT: (4,5)
LEARNING RATE: 0.1

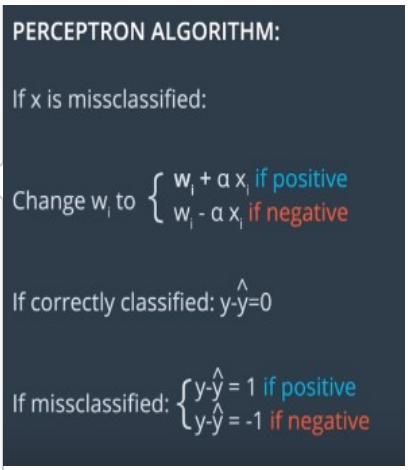
	3	4	-10
−	0.4	0.5	0.1
	2.6	3.5	-10.1

NEW LINE
$2.6x_1 + 3.5x_2 - 10.1 = 0$

3.10 Perceptron Algorithm

```
1. Start with random weights: w1,..., wn, b
2. For Every misclassified point(x1,...., xn):
    2.1 If the prediction = 0
        - For i = 1...n
            - change w_i + alpha*x_i
        change b to b + alpha

    2.2 If the prediction = 1
```

```
        - For i = 1....n
            -change w_i - alpha*x_i
        -change b to b - alpha
```




```
import numpy as np
# Setting the random seed, feel free to change it and
see different solutions.
np.random.seed(42)

def stepFunction(t):
    if t >= 0:
        return 1
    return 0

def prediction(X, W, b):
    return stepFunction((np.matmul(X,W)+b)[0])

# TODO: Fill in the code below to implement the
perceptron trick.
# The function should receive as inputs the data X, the
labels y,
# the weights W (as an array), and the bias b,
# update the weights and bias W, b, according to the
perceptron algorithm,
# and return W and b.
def perceptronStep(X, y, W, b, learn_rate = 0.01):
    # Fill in code
    for i in range(len(X)):
        y_hat = prediction(X[i], W, b)

        if y[i]-y_hat == 1:
            W[0] += X[i][0] * learn_rate
            W[1] += X[i][1] * learn_rate
            b += learn_rate

        elif y[i]-y_hat == -1:
            W[0] -= X[i][0] * learn_rate
            W[1] -= X[i][1] * learn_rate
            b -= learn_rate

    return W, b

# This function runs the perceptron algorithm repeatedly
on the dataset,
# and returns a few of the boundary lines obtained in
the iterations,
# for plotting purposes.
# Feel free to play with the learning rate and the
num_epochs,
# and see your results plotted below.
def trainPerceptronAlgorithm(X, y, learn_rate = 0.01,
```

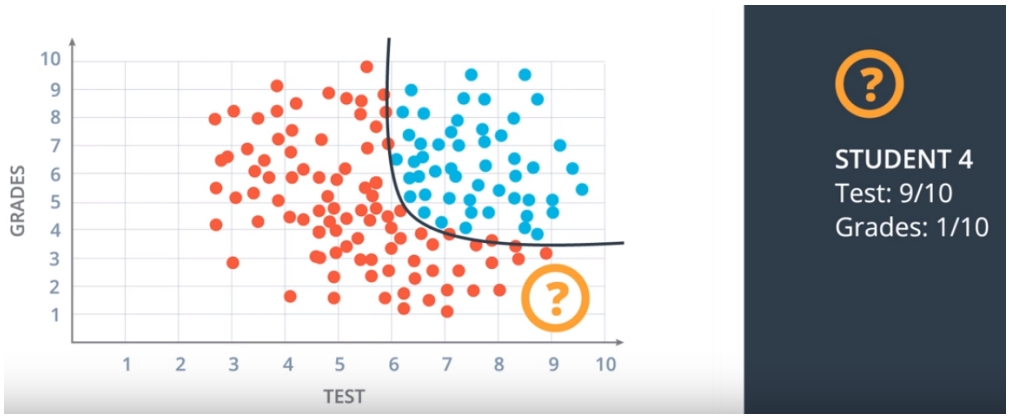
```
    num_epochs = 25):
        x_min, x_max = min(X.T[0]), max(X.T[0])
        y_min, y_max = min(X.T[1]), max(X.T[1])
        W = np.array(np.random.rand(2,1))
        b = np.random.rand(1)[0] + x_max
        # These are the solution lines that get plotted
    below.
        boundary_lines = []
        for i in range(num_epochs):
            # In each epoch, we apply the perceptron step.
            W, b = perceptronStep(X, y, W, b, learn_rate)
            boundary_lines.append((-W[0]/W[1], -b/W[1]))

        return boundary_lines
```

3.11 Non Linear Regions

Let's say that we have a student who get 9 on test and 1 on grades that mean according to that model he should be in the accepted area.

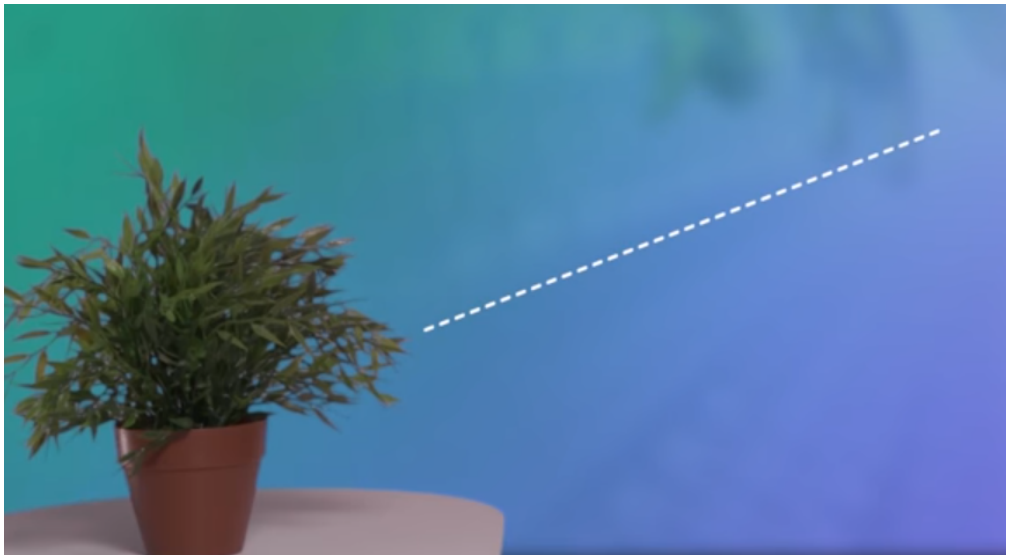
But that's wont be reasonable because he should what ever he got in the test if the grade is low, he should,'t be accepted, so the data should be more like that

But now the data cannot be separated by a line, we need more complexity to fit that data. we might fit the data with the curve and that's mean the perceptron algorithm won't work this time and we need to redefine our perceptron algorithm in away that it will generalize to other types of curves



3.12 Error Functions

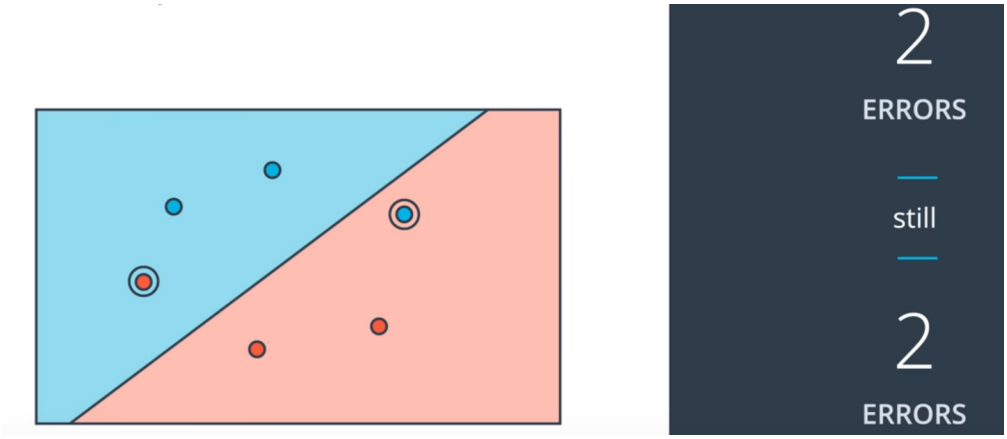
The Error function is used to decrease the distance between function and the target



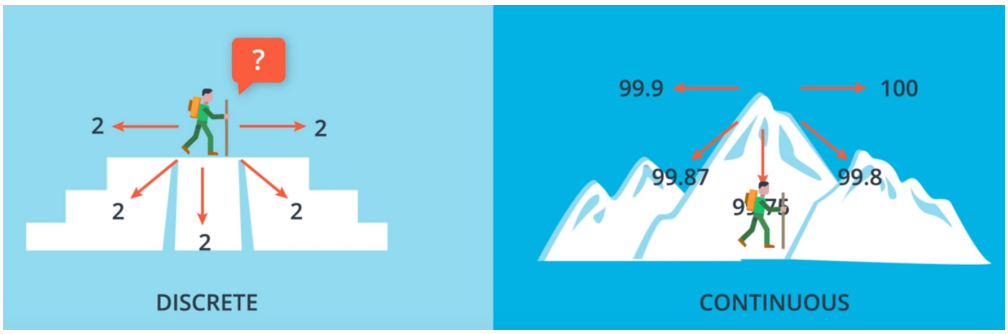
3.13 Log-loss Error Function

We use these error functions to decrease the error to reach the best minimum or the global minimum which will help the function to do well on the data and get the nearest right results

We take so tiny steps to decrease the error and the reason for that is calculus ,because we take the derivatives to calculate it. But when we use tiny steps we cannot go further.



If we try to descend from Aztec pyramid with flat steps, when we look at every dimention we found out that we cannot go down and get confused. But with mount Errorest when we look in every dimension we find very small variation in height and we can detect in what direction we can decrease the most.

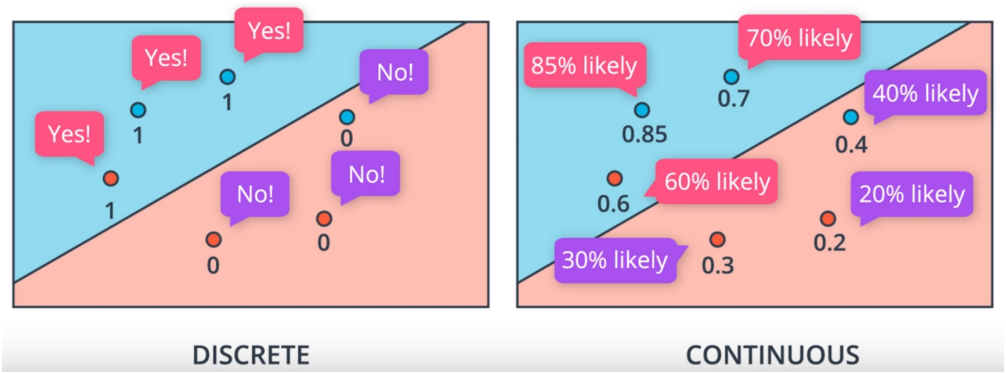


Gradient Descent Terms:-

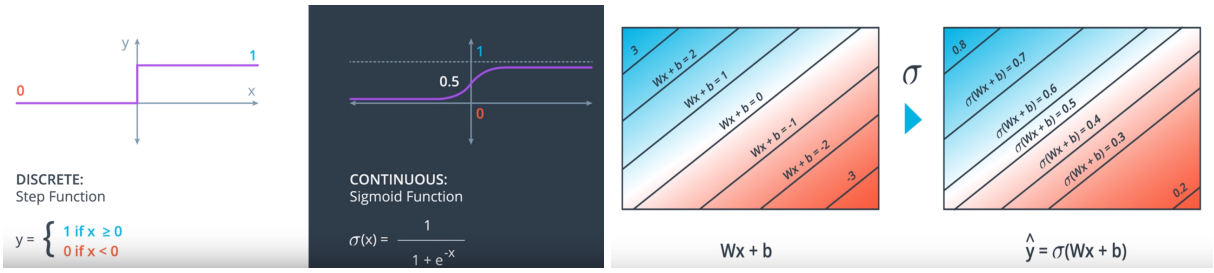
- The values should be continous.
- Error Function should differentiable.

3.14 Discrete vs Continous

The predictions of disceret values will be 0,1 but for continous values will be probability as long as the probability is high it means the values in the positive area and if it low means in negative area.



So to move from disceret predictions to continous, change the activation function from "Step Function" to "Sigmoid Function"



We apply this to Neural Networks by changing the perceptron of "Step Function" to "Sigmoid Function" as New Activation Function.