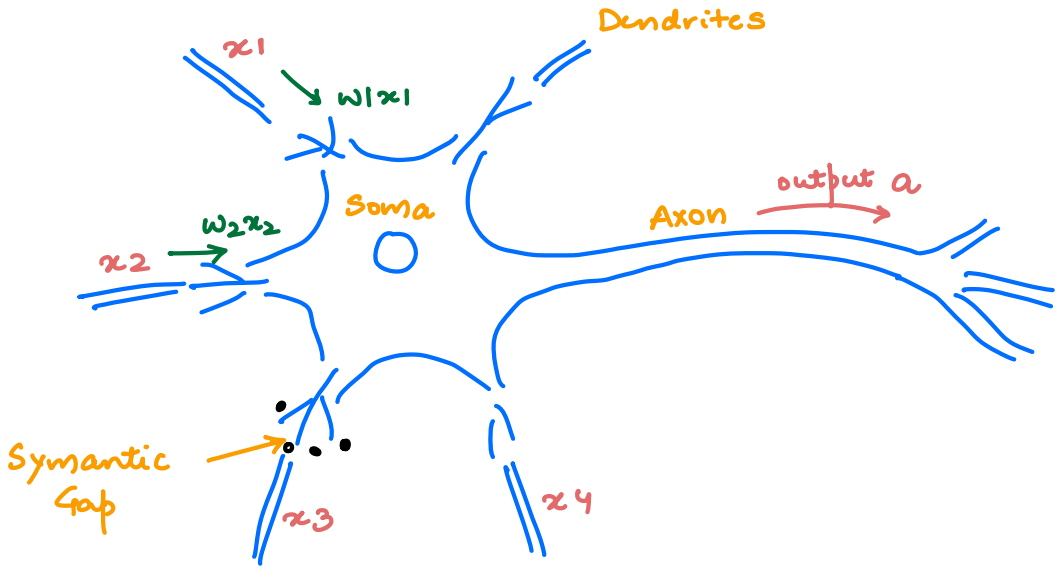
ARTIFICIAL V/S BIOLOGICAL NEURONS

In this lecture we will understand how the idea of neural networks is inspired from working of human brain. We will also talk about similarities b/w a biological neuron and an artificial neuron.

Neuron is the basic unit of a neural network. We will be building a network which will do various tasks like predicting language, language modelling, classifying images. Google Allo, Alexa all use deep learning which is based on neural network.

Firstly lets understand how a neural network is related to a biological neuron.



Biological Neuron

- In brain you have collection of neurons and each neuron looks like this ..... and you have dendrites which brings

input from other neurons to current neuron. Lets say each dendrite is bringing input x_1, x_2, x_3 and x_4.

- This input goes to the processing unit called as **Soma**, here neuron does some processing of the information.
- In b/w there is **Synaptic gap** which basically modifies the signals. Signals are basically in the form of electrical and chemical impulses. Lets say the original input was x_1 but the input which goes to Soma will be $w_1 x_1$ (i.e. weighted input). Soma will calculate sum of all inputs $z = \sum w^i x^i$
- Some neurons will fire and some won't. Firing of neuron depends on activation function. When $z = \sum w^i x^i >$ threshold value then we can say neuron will fire.

output is denoted by a

$$z = w_i x_i$$

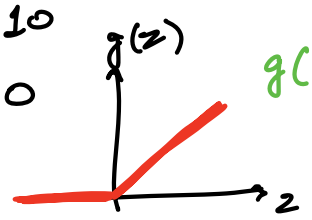
$$a = g(z)$$

g is a threshold fx^n which is also called as activation fx^n.

If z is greater than some threshold then neuron is going to fire otherwise neuron will not fire. In practice g can be a sigmoid fx^n but nowadays we use very interesting fx^n called as Relu (Rectified Linear Unit)

$g(10) = 10$

$g(-5) = 0$



$$g(z) = \begin{cases} z & z \geq 0 \\ 0 & z < 0 \end{cases}$$

Neuron fires

Neuron Doesn't fire

We should add a bias term to z. $z = w_i x_i + b$

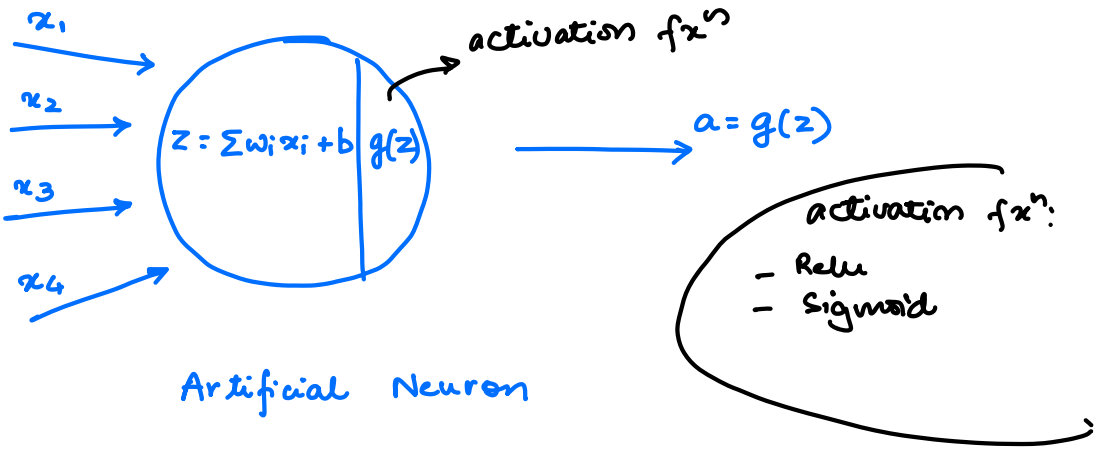
If we want that a neuron should fire when the value is -10, we will keep bias as +10.

$$z = w_i x_i + b$$

$w_i x_i$: -10, b: +10

z: 0

Even if $w_i x_i$ is -10, still neuron will fire.



Artificial Neuron

w_i is the weight that we will learn using algo like Gradient Descent. This is very similar to Logistic Regression if your activation fxn is a sigmoid fxn.

Resemblance b/w Biological and Artificial Neuron:

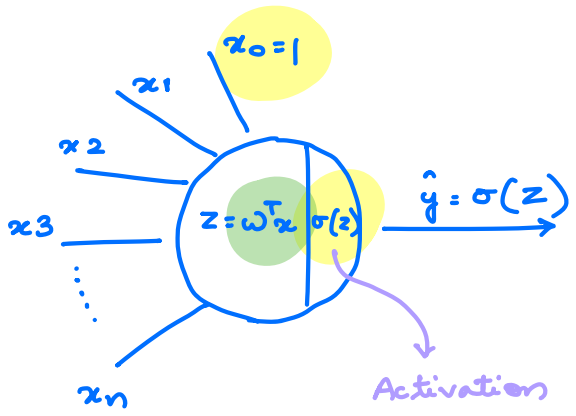
Biological	Artificial
Dendrites	Inputs

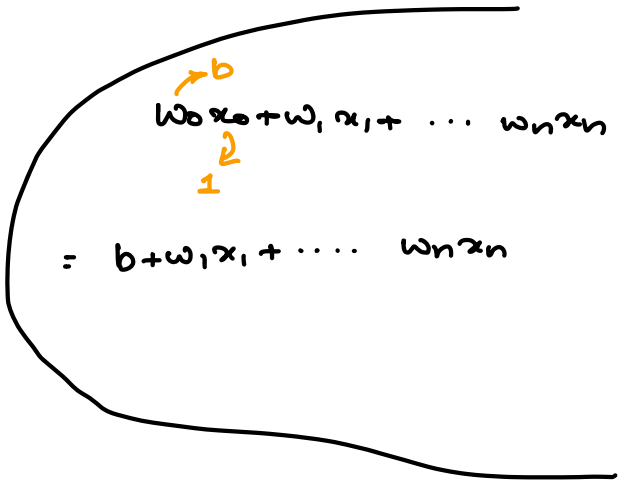
x_1, x_2, x_3 → features

Synaptic Gap	weights
Axon	output
Soma	Activation

HOW DOES AN ARTIFICIAL NEURON LEARN?

In this lecture we will see what is a Perceptron. Perceptron is a single layer neural network. We will be learning about the loss function and later on we will derive weight update rule.

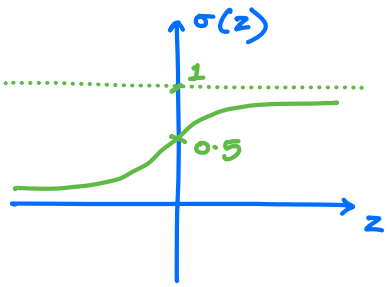




$$z = w^T x = \begin{bmatrix} w_0 & w_1 & w_2 & \cdots & w_n \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = w_0x_0 + w_1x_1 + \cdots + w_nx_n$$

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

o/p is any no. b/w 0 and 1



Let us say we are doing a classification problems in which we want to predict if given image is of dog or cat. (dog → 0, cat → 1)

features: pixel values

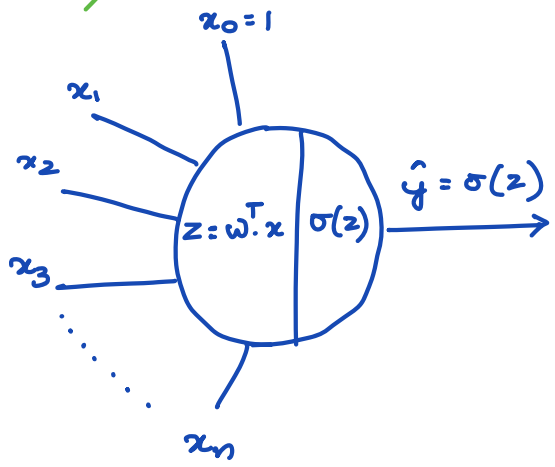
 → 0 or 1 ?

$\hat{y}$ lies in range 0 and 1.

$w_1x_1 + w_2x_2 \cdots\cdots w_{100}x_{100} = z \longrightarrow \sigma(z) \longrightarrow 0$ or 1

If $\hat{y} = (0.8) \approx 1$ (Round off to 1)
$\hat{y} = (0.3) \approx 0$ (Round off to 0)
} $\hat{y}$ tells which class a given input belongs to.

Perceptron will act as a linear classifier. It can do Binary Classification)

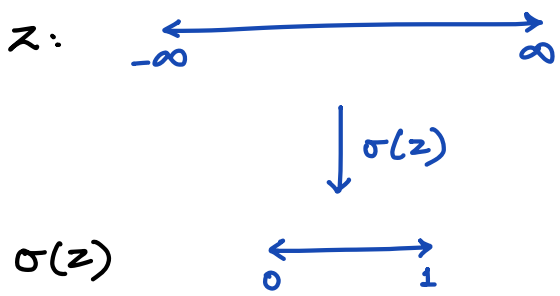


Basic Neural Network with only 1 node

Goal of learning Algorithm is to learn parameters w, if you get data of type 1 then output should be close to 1 and if you get data of type 0 then output should be close to 0.

If input is very large, lets say $z = 100$ then $\sigma(z) = 1$
$z = -200$ then $\sigma(z) = 0$

Activation fx^n is compressing your number line in the range 0 to 1.



$\sigma(z)$ tells the probability a given input belongs to which class.

Lets say $\sigma(z) = 0.7$, you are 70% confident that it belongs to class 1, you are 30% confident that it belongs to class 0.

$$P(y=1) \longrightarrow \sigma(z)$$

$$P(y=0) \longrightarrow 1-\sigma(z)$$

$$z = w^T \cdot x$$

This is how we can make predictions using a simple perceptron.

How to train a perceptron ?

In every ML Algo, we do the following:

- Model (Single Node of Perceptron, when we combine multiple such nodes we get a neural network which is called as multi layer perceptron)

- Loss

Algo predicts probability, with what probability it belongs to class Dog.

$P(y=1)$: $\hat{y}$: 0.7 0.8 0.8 0.3 0.4

→ with 0.3 probability it belongs to class Dog, it belongs to class cat with probability 0.7

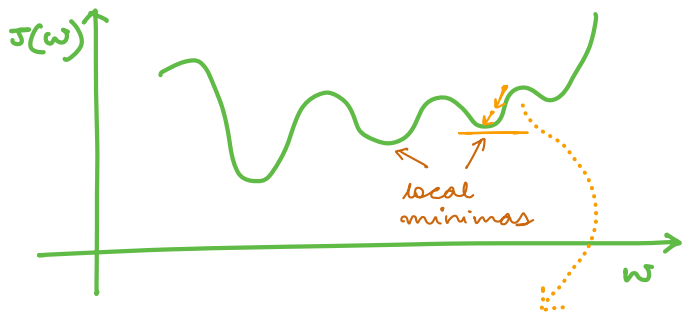
y : 1 1 1 0 0

$$J(w) \nearrow \quad MSE = \frac{1}{m} \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)} \right)^2$$

$$\hat{y}^{(i)} = \sigma(z) = \sigma(w^T . x)$$

$\hat{y}^{(i)}$ is a function of w.

↳ Problem with this kind of loss function is it is a **non-convex function.**



If you start from this point and use gradient descent then you will be stuck in local minima. There are multiple local minimas and you will stuck inside it.

To overcome this difficulty we use another kind of loss called as log loss.

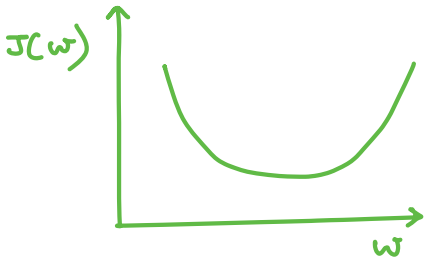
Log loss / Binary Cross Entropy

↳ Proof for this is already covered in Maximum Likelihood Estimation for Logistic Regression.

$$-\sum_{i=1}^{m} \left(y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log (1-\hat{y}^{(i)}) \right)$$

↳ It is a **convex fx^n**

↳ only 1 local and global minima



- if $y^{(i)} = 0$

  then loss $fx^n = \log(1 - \hat{y}^{(i)})$

- if $y^{(i)} = 1$

  then loss $fx^n = \log \hat{y}^{(i)}$ → if predictions are not correct then we add log of misclassification.

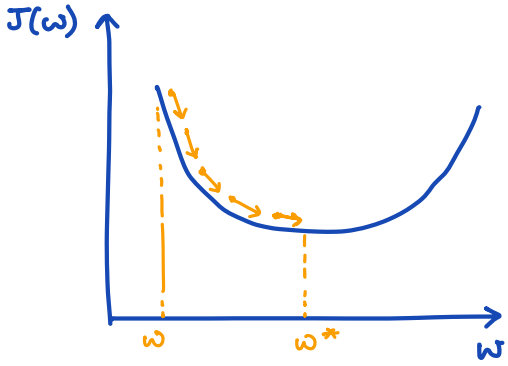
- if $y^{(i)} = 0$ and $\hat{y}^{(i)} = 0$

  then loss $fx^n = 0$ → that means when actual values matches with predicted value then you are adding 0 to loss.

- if $y^{(i)} = 1$ and $\hat{y}^{(i)} = 1$

  then loss $fx^n = 0$

- Weight update Rule



Start with some random w and update w as:

$$w = w - \eta \left(\frac{\partial J}{\partial w} \right)$$

-ve

+ve

} Gradient Update Rule

$$J(w) = -\sum_{i=1}^{m} y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log (1 - \hat{y}^{(i)})$$

$$\hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)})$$

$J(w)$ is a fx^n of $\hat{y}^{(i)}$

$\hat{y}^{(i)}$ is a fx^n of $w^T x^{(i)}$

GRADIENT DESCENT UPDATE

In the last lecture we talked about loss function to train a perceptron and how perceptron acts as a binary classifier. In this video our aim is to derive weight update rule that will help us to find optimal set of parameters w for the perceptron and we will use gradient descent update rule to update the parameters.

$$J(w) = -\sum_{i=1}^{m} y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log (1-\hat{y}^{(i)})$$

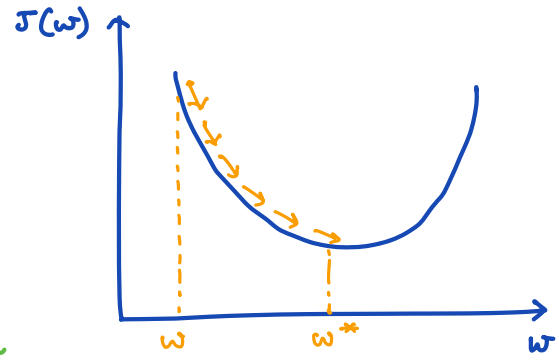
Loss function

Goal is to learn w

$J(w)$ is a convex fxn

we start with any random w and we want to end up at some optimal w^*

This can be done by using Gradient update rule which repeatedly decreases our loss in the direction of reducing gradient.



$w = [w_0 \; w_1 \; w_2 \; \ldots\ldots \; w_n]$ n = no. of features

$$\frac{\partial J(w)}{\partial w} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w}$$

(Chain Rule)

$\hat{y} = \sigma(z)$

$z = w^T \cdot x$

for 1 example $= -\left(\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}}\right)$

later we will add i

$$J(w) = -\sum_{i=1}^{m} y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log (1-\hat{y}^{(i)})$$

$$\hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)})$$

$\frac{\partial \hat{y}}{\partial z}$?

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

$$\frac{\partial \hat{y}}{\partial z} = \frac{e^{-z}}{(1+e^{-z})^2}$$

$$\frac{\partial \hat{y}}{\partial z} = \left(\frac{1}{1+e^{-z}}\right)\left(1 - \frac{1}{1+e^{-z}}\right)$$

$$\frac{\partial \hat{y}}{\partial z} = \sigma(z)\,(1-\sigma(z))$$

$$\hat{y}' = \sigma(z)\,(1-\sigma(z))$$

$$\hat{y}' = \hat{y}\,(1-\hat{y})$$

$\hat{y} = \sigma(z)$

$\hat{y}' = \sigma'(z)$

$\frac{\partial z}{\partial w}$?

$$z = w^T x$$

$$z = w_0 + w_1 x_1 + \cdots\cdots w_i x_i + \cdots + w_n x_n$$

$$\frac{\partial z}{\partial w_i} = x_i$$

$$\frac{\partial J(w)}{\partial w_i} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_i}$$

$$= \left(-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right)\left(\hat{y}(1-\hat{y})\right) x_i$$

$$= \frac{(-y + y\cdot\hat{y} + \hat{y} - \hat{y}\cdot y)\,(\hat{y}(1-\hat{y}))}{\hat{y}(1-\hat{y})}\, x_i$$

$$= (\hat{y} - y)\, x_i$$

$$\frac{\partial J(w)}{\partial w_i} = \left(\sigma(w^T \cdot x) - y\right) x_i$$

→ ith feature for given 1 example

$$w_j = w_j - \eta \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right) x_j^{(i)}$$

→ update the jth gradient.

Single Perceptron + sigmoid : Logistic Regression

PERCEPTRON IMPLEMENTATION

In this video we will implement a perceptron which is a single layer neural network. It acts as a linear classifier. We will use binary cross entropy as a loss function and gradient descent optimizer.

Learning Goals ?

- How to implement Perceptron.
- We won't use any for loop, we will see how vectorization in python works.
- What happens when you use non linear dataset and a linear classifier like perceptron.

CODE:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
```

Generating Data

```
X, Y = make_blobs ?
```

→ it returns gaussian blobs

```
X, Y = make_blobs (n_samples = 500, centers = 2, n_features = 2,
                   random_state = 10)
print (X.shape, Y.shape)
```

→ (500, 2) (500,)

plt.style.use("seaborn");

plt.scatter(x[:,0], x[:,1], c=Y, cmap = plt.cm.Accent)

plt.show()

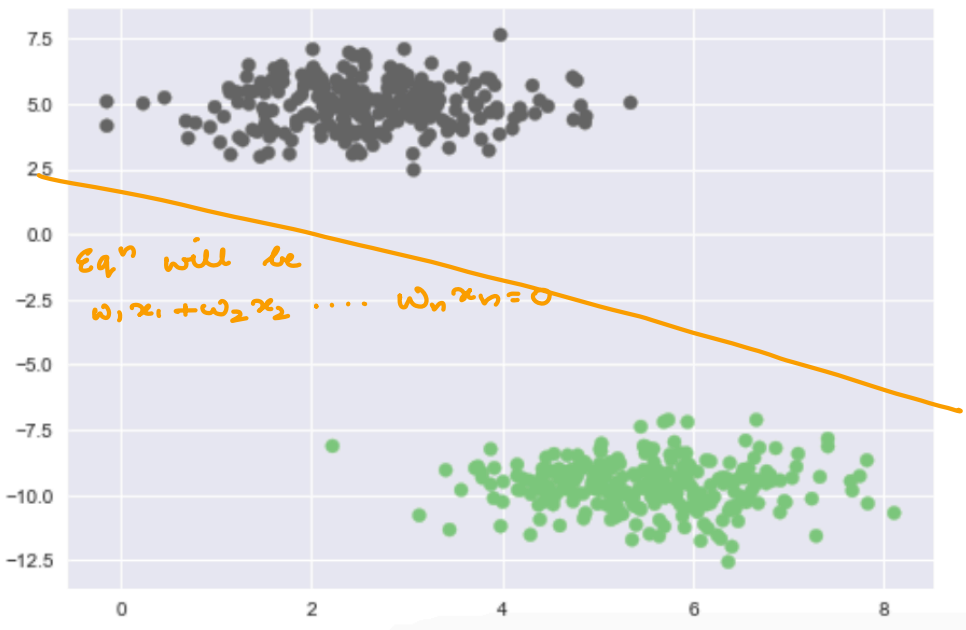
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
```

Generating Data

```
X,Y = make_blobs(n_samples=500, centers=2, n_features=2, random_state=10)
print(X.shape, Y.shape)
```

```
(500, 2) (500,)
```

```
plt.style.use("seaborn")
plt.scatter(X[:,0], X[:,1], c=Y, cmap=plt.cm.Accent)
plt.show()
```



→ it is linearly separable data.

Goal of perceptron learning algo is to figure out a boundary which separates data into 2 classes.

Model and Helper functions

def sigmoid(z):

    return (1.0)/(1+np.exp(-z))

z = np.array([1,2,3,4,5])

sigmoid(z)

you will get an array.
This kind of functionality is called broadcasting
Sigmoid fxn is now applied on every element of array.

$$\sigma\begin{bmatrix}1\\2\\3\\4\\5\end{bmatrix} = \begin{matrix}\sigma(1)\\ \sigma(2)\\ \sigma(3)\\ \sigma(4)\\ \sigma(5)\end{matrix}$$

It is possible only in numpy array.
Numpy does it bcz of a technique called as broadcasting.

Model and Helper Functions

```
def sigmoid(z) :
    return (1.0)/(1+np.exp(-z))
```

```
z = np.array([1,2,3,4,5])
sigmoid(z)
```

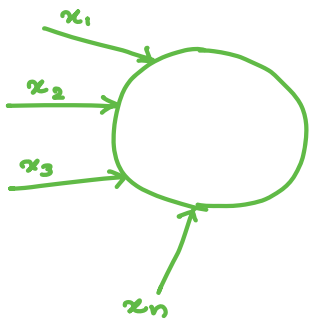
```
array([0.73105858, 0.88079708, 0.95257413, 0.98201379, 0.99330715])
```

Implement Perceptron Learning Algorithm

- Learn the weights
- Reduce the loss
- Make the predictions

$w = [w_0\ w_1\ \dots\ w_n]$ $n \times 1$

$x = \begin{bmatrix}—x^1—\\ —x^2—\\ —x^m—\end{bmatrix}$ $m \times n$



$m \times 1$ output

```
def predict (x, weights) :
    z = np.dot (x, weights)
    predictions = sigmoid(z)
    return predictions
```

$X_{m \times (n+1)}$ matrix

$w_{n \times 1}$ vector

```
def loss (x, y, weights):
    """ Binary Cross Entropy """
    y_ = predict (x, weights)
    cost = np.mean (-y * np.log (y_) - (1-y) * np.log (1- y_))
    return cost
```

$$\frac{-1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log (1-\hat{y}^{(i)}) \right)$$

```
def update ( x , y , weights , learning_rate ):
    """ Perform weight updates for 1 epoch """

    y_ = predict (x, weights)
    dw = np.dot ( x.T , y_ - y )

    m = x.shape[0]
    weights = weights - learning_rate *dw / (float(m))
    return weights
```

$$w_j := w_j - \eta \cdot \frac{\partial J}{\partial w_j}$$

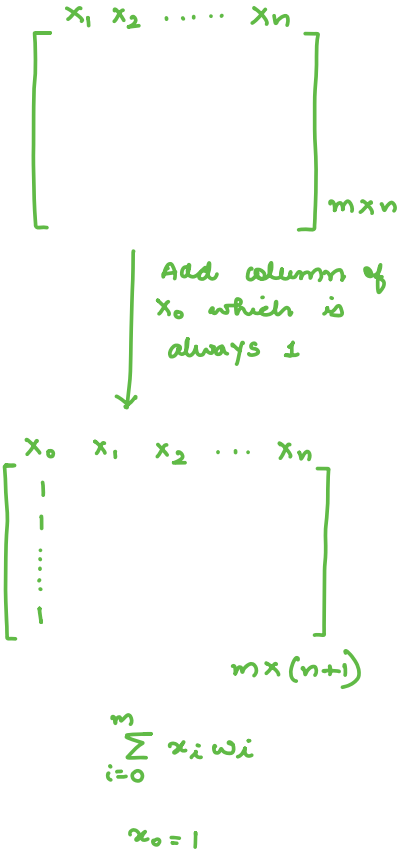
$$\frac{\partial J}{\partial w_j} = (\hat{y} - y)\, x_j$$

$$X = \begin{bmatrix} — & x^1 & — \\ — & x^2 & — \\ — & x^3 & — \\ & \vdots & \\ — & x^m & — \end{bmatrix}_{m \times n} \qquad \begin{matrix} \hat{y} - y \\ \begin{bmatrix} \\ \\ \\ \end{bmatrix}_{m \times 1} \end{matrix}$$

$$X^T = \begin{bmatrix} | & | & | & | \\ x^1 & x^2 & x^i & x^m \\ | & | & | & | \end{bmatrix}_{n \times m} \qquad \begin{matrix} \hat{y} - y \\ \begin{bmatrix} \\ \hat{y}^i - y^i \\ \\ \end{bmatrix}_{m \times 1} \end{matrix}$$

$$(\hat{y}^{(i)} - y^{(i)}) x_j^{(i)}$$

```
def train (x, y, learning_rate = 0.5, max epochs = 100):

    # Modify input to handle bias term
    ones = np.ones ((x.shape[0], 1))
    x = np.hstack ((ones, x))

    # Init weights 0
    weights = np.zeros (x.shape [1])

    # Iterate over all epochs and make
      updates
    for epoch in range (max epochs):
        weights = update (x, y, weights,
                          learning_rate)

    after every 10 epochs print the progress
    if epoch %.10 == 0:
        l = loss (x, y, weights)
        print ("epoch %.d LOSS %.4f" %. (epoch, l))

    return weights

train (x, y)
```



$$\sum_{i=0}^{m} x_i w_i$$

$x_0 = 1$

Implement Perceptron Learning Algorithm

- Learn the weights
- Reduce the loss
- Make the pRedictions

```
def predict(X,weights) :
    """X -> m X (n+1) matrix , w -> n X 1 vector"""
    z = np.dot(X, weights)
    predictions = sigmoid(z)
    return predictions

def loss(X,Y,weights) :
    """Binary Cross Entropy"""
    Y_ = predict(X, weights)
    cost = np.mean(-Y*np.log(Y_) - (1-Y)*np.log(1-Y_))
    return cost

def update(X, Y, weights, learning_rate) :
    """Perform weight updates for 1 epoch"""
    Y_ = predict(X, weights)
    dw = np.dot(X.T, Y_-Y)

    m = X.shape[0]
    weights = weights - learning_rate*dw/(float(m))
    return weights

def train(X, Y, learning_rate=0.5, maxEpochs=100) :

    #Modify the input to handle the bias term
    ones = np.ones((X.shape[0],1))
    X = np.hstack((ones,X))

    #Init Weights 0
    weights = np.zeros(X.shape[1]) # n+1 entries

    #Iterate over all epochs and make updates
    for epoch in range(maxEpochs) :
        weights = update(X, Y, weights, learning_rate)

        if epoch % 10 == 0 :
            l = loss(X, Y, weights)
            print("Epoch %d Loss %.4f"%(epoch,l))

    return weights
```

```
train(X,Y)
```

```
Epoch 0 Loss 0.0006
Epoch 10 Loss 0.0005
Epoch 20 Loss 0.0005
Epoch 30 Loss 0.0005
Epoch 40 Loss 0.0005
Epoch 50 Loss 0.0004
Epoch 60 Loss 0.0004
Epoch 70 Loss 0.0004
Epoch 80 Loss 0.0004
Epoch 90 Loss 0.0004
```

Loss is Reducing.

```
array([ 0.02204952, -0.30768518,  1.90003958])
```

w_0 w_1 w_2

Weights learnt by your classifier. For n features, we will have n+1 weights.

weights = train(X, Y, maxEpochs=500)

```
weights = train(X, Y, maxEpochs=500)
```

```
Epoch 0 Loss 0.0006
Epoch 10 Loss 0.0005
Epoch 20 Loss 0.0005
Epoch 30 Loss 0.0005
Epoch 40 Loss 0.0005
Epoch 50 Loss 0.0004
Epoch 60 Loss 0.0004
Epoch 70 Loss 0.0004
Epoch 80 Loss 0.0004
Epoch 90 Loss 0.0004
Epoch 100 Loss 0.0004
Epoch 110 Loss 0.0003
Epoch 120 Loss 0.0003
Epoch 130 Loss 0.0003
Epoch 140 Loss 0.0003
Epoch 150 Loss 0.0003
Epoch 160 Loss 0.0003
Epoch 170 Loss 0.0003
Epoch 180 Loss 0.0003
Epoch 190 Loss 0.0003
Epoch 200 Loss 0.0003
Epoch 210 Loss 0.0003
Epoch 220 Loss 0.0002
Epoch 230 Loss 0.0002
Epoch 240 Loss 0.0002
Epoch 250 Loss 0.0002
Epoch 260 Loss 0.0002
Epoch 270 Loss 0.0002
Epoch 280 Loss 0.0002
Epoch 290 Loss 0.0002
Epoch 300 Loss 0.0002
Epoch 310 Loss 0.0002
Epoch 320 Loss 0.0002
Epoch 330 Loss 0.0002
Epoch 340 Loss 0.0002
Epoch 350 Loss 0.0002
Epoch 360 Loss 0.0002
Epoch 370 Loss 0.0002
Epoch 380 Loss 0.0002
Epoch 390 Loss 0.0002
Epoch 400 Loss 0.0002
Epoch 410 Loss 0.0002
Epoch 420 Loss 0.0002
Epoch 430 Loss 0.0002
Epoch 440 Loss 0.0002
Epoch 450 Loss 0.0002
Epoch 460 Loss 0.0002
Epoch 470 Loss 0.0002
Epoch 480 Loss 0.0001
Epoch 490 Loss 0.0001
```

loss is continuously reducing

→ loss is close to 0

VISUALISING DECISION SURFACE

We are building a simple neural network by training one perceptron. We will see how to make predictions for all data points and also visualise the hypothesis generated by our algo. We will run this perceptron over a non linear dataset and will see the results.

Perceptron Implementation - Part II

- Make Predictions
- Visualise Decision Surface
- Linear vs Non-linear Classification

```
def getPredictions(x_Test, weights, labels=True):

    if x_Test.shape[1] != weights.shape[0]:
        ones = np.ones((x_Test.shape[0], 1))
        x_Test = np.hstack((ones, x_Test))

    probs = predict(x_Test, weights)

    if not labels:
        return probs

    else:
        labels = np.zeros(probs.shape)
        labels[probs >= 0.5] = 1
        return labels
```

if test data doesnot have column of 1's appended then we are going to add 0.

$(m \times n)$ $(n+1)$

$w_0, w_1, \ldots, w_n$

$X \cdot W$

$m \times (n+1)$ $(n+1) \times 1$

output of algo is going to be a vector

$$\begin{bmatrix} \\ x \end{bmatrix} = \begin{bmatrix} \text{pred} \\ 0.8 \\ 0.7 \\ 0.6 \\ \vdots \end{bmatrix}$$

Vectorization is being used.

All the places where probability is >0.5 make label as 1

a = np.zeros((5,5))

a[2,3] = 10

print(a)

a[a>0] = 20

print(a)

Perceptron Implementation - Part II

- Make Predictions
- Visualise Decision Surface
- Linear vs Non-Linear Classification

```
def getPredictions(X_Test, weights, labels=True) :

    if X_Test.shape[1] != weights.shape[0] :
        ones = np.ones((X_Test.shape[0], 1))
        X_Test = np.hstack((ones, X_Test))

    probs = predict(X_Test, weights)

    if not labels :
        return probs
    else :
        labels = np.zeros(probs.shape)
        labels[probs >= 0.5] = 1
        return labels
```

```
a = np.zeros((5,5))
a[2,3] = 10
print(a)

a[a>0] = 20
print(a)
```

```
[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0. 10.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]
[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0. 20.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]
```

weights

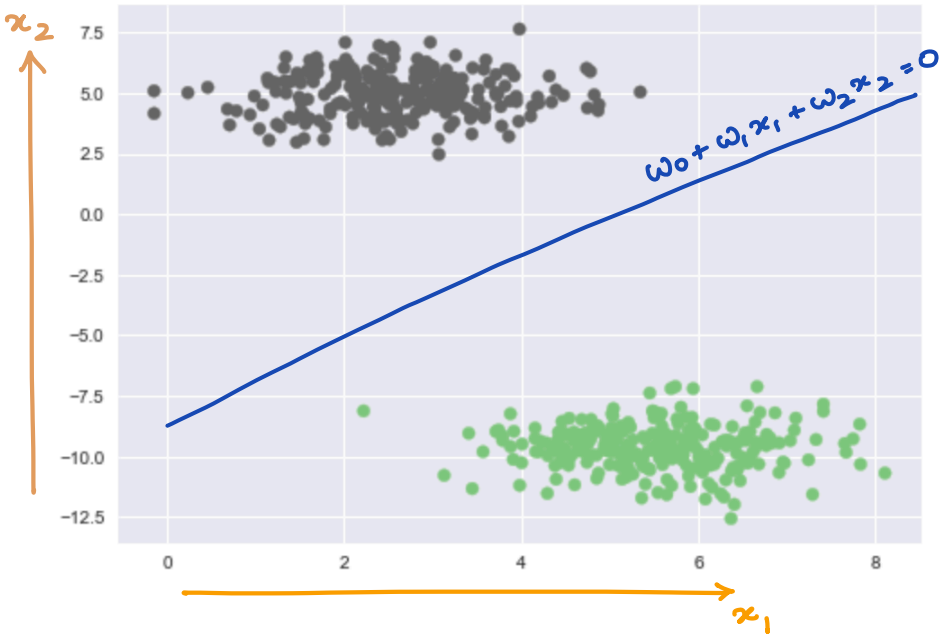
plt.scatter(x[:,0], x[:,1], c=y, cmap=plt.cm.Accent)

plt.show()

```
weights
```
w_0 w_1 w_2

```
array([ 0.0656045 , -0.19220497,  2.05528664])
```

```
plt.scatter(X[:,0], X[:,1], c=Y, cmap=plt.cm.Accent)
plt.show()
```



$$x_2 = -\frac{(w_0 + w_1 x_1)}{w_2}$$

If we plot x_2 v/s x_1 using this formula we will get the line.

```
x1 = np.linspace(-8, 2, 10)
x2 = -(weights[0] + weights[1] * x1) / weights[2]

plt.scatter(x[:,0], x[:,1], c=y, cmap=plt.cm.Accent)
plt.plot(x1, x2, c='red')
plt.show()
```

```
x1 = np.linspace(-1,10,10)
x2 = -(weights[0] + weights[1]*x1) / weights[2]
```

```
plt.scatter(X[:,0], X[:,1], c=Y, cmap=plt.cm.Accent)
plt.plot(x1, x2, c='red')
plt.show()
```

→ we are able to separate the data

Lets try with different value of random_state

X, Y = make_blobs (n_samples = 500, centers = 2, n_features = 2, random_state = 1)

```
X,Y = make_blobs(n_samples=500, centers=2, n_features=2, random_state=1)
print(X.shape, Y.shape)
```

```
x1 = np.linspace(-15,5,10)
x2 = -(weights[0] + weights[1]*x1) / weights[2]
```

```
plt.scatter(X[:,0], X[:,1], c=Y, cmap=plt.cm.Accent)
plt.plot(x1, x2, c='red')
plt.show()
```

find the accuracy

Y_ = getPredictions (X, weights , labels = True)

print (Y_)

```
#Find the accuracy
Y_ = getPredictions(X, weights, labels=True)
print(Y_)
```

```
[1. 0. 1. 0. 1. 0. 0. 1. 0. 1. 1. 1. 1. 0. 0. 1. 0. 1. 0. 1. 1. 1. 0. 0.
 1. 1. 0. 0. 0. 1. 0. 0. 1. 1. 1. 1. 0. 0. 1. 0. 1. 1. 0. 1. 1. 1. 0. 1.
 0. 0. 0. 1. 1. 1. 0. 1. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0. 1. 1. 0.
 1. 1. 1. 0. 1. 1. 1. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 1. 0. 1.
 0. 1. 1. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1.
 1. 0. 1. 1. 1. 0. 0. 1. 1. 0. 0. 0. 0. 1. 1. 1. 0. 1. 0. 0. 1. 0. 1. 0.
 0. 0. 1. 0. 1. 1. 1. 0. 0. 1. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1.
 0. 1. 0. 0. 0. 0. 0. 1. 0. 1. 1. 1. 0. 0. 1. 1. 1. 1. 1. 0. 1. 0. 0. 1.
 1. 1. 0. 1. 0. 1. 0. 1. 0. 1. 1. 1. 1. 0. 1. 1. 0. 1. 0. 1. 1. 1. 1. 0.
 0. 0. 1. 1. 0. 1. 0. 0. 0. 1. 0. 0. 1. 0. 1. 0. 0. 0. 0. 1. 0. 1. 1. 0.
 1. 1. 1. 1. 1. 0. 1. 0. 1. 0. 0. 1. 1. 0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 0.
 1. 1. 1. 0. 0. 0. 1. 0. 0. 1. 0. 0. 1. 0. 1. 0. 1. 1. 0. 1. 0. 0. 1. 0.
 0. 0. 1. 1. 1. 1. 0. 1. 0. 1. 1. 1. 1. 1. 0. 1. 1. 0. 1. 1. 0. 0. 1. 1.
 1. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 1. 0. 0. 0. 1. 1. 1. 0. 0. 1. 1. 0. 0.
 1. 1. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 0. 0. 1. 1. 0. 0. 0. 1. 0.
 0. 1. 0. 0. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 0. 1. 0. 1. 0. 0. 0. 0. 1. 0.
 1. 1. 0. 0. 0. 1. 1. 0. 1. 0. 0. 0. 0. 1. 0. 1. 0. 1. 1. 1. 1. 1. 1. 0.
 0. 1. 0. 1. 0. 0. 1. 0. 0. 1. 0. 0. 1. 0. 1. 1. 0. 0. 0. 1. 0. 1. 1. 0.
 1. 1. 0. 1. 1. 1. 0. 0. 1. 1. 1. 1. 0. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0.
 1. 1. 0. 0. 1. 0. 1. 0. 0. 0. 0. 1. 0. 1. 0. 0. 1. 1. 0. 1. 0. 1. 1. 0.
 0. 1. 0. 1. 1. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1. 1. 1. 0. 1.]
```

```
Y_ == Y
```

```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
```

element wise comparison

500

```
training_acc = np.sum(Y_==Y)/Y.shape[0]
print(training_acc*100)
```

```
100.0
```

Training Accuracy is 100%
All predictions are correct

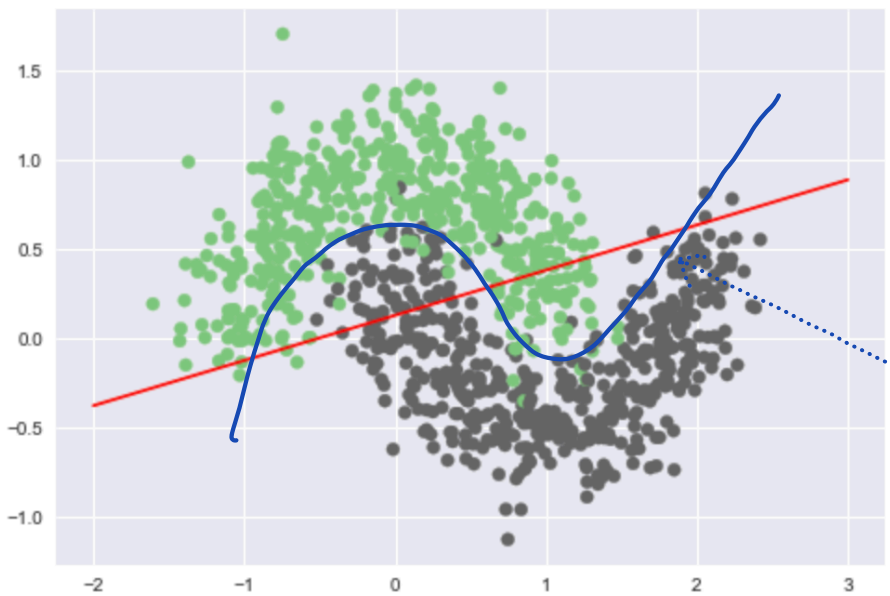
what if we use non-linear data ?

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs, make_moons
```

```
# X,Y = make_blobs(n_samples=500, centers=2, n_features=2, random_state=1)
X,Y = make_moons(n_samples=1000, shuffle=True, noise=0.2, random_state=1)
print(X.shape, Y.shape)
```

```
x1 = np.linspace(-2,3,10)
x2 = -(weights[0] + weights[1]*x1) / weights[2]
```

```
plt.scatter(X[:,0], X[:,1], c=Y, cmap=plt.cm.Accent)
plt.plot(x1, x2, c='red')
plt.show()
```



Now, not able to learn a very good boundary bcz data was not linear.

We need to learn some other hypothesis which should be able to separate data like this.
This kind of boundary can be learnt by MLP (Multi layer Perceptron)

```
training_acc = np.sum(Y_==Y)/Y.shape[0]
print(training_acc*100)
```

```
86.8
```

now accuracy is only 86.8%
Some points are not classified correctly.