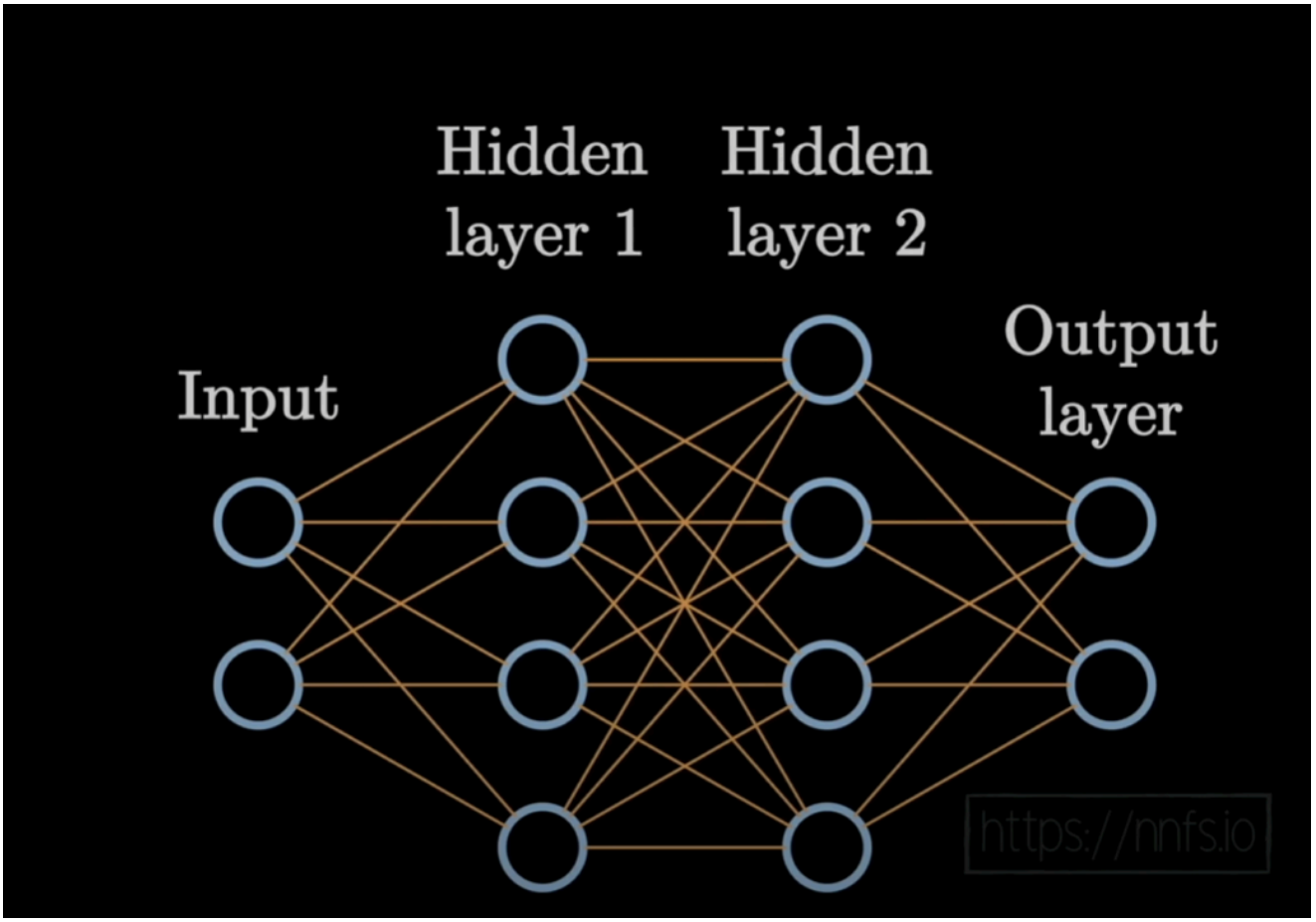
Neural Network - Data Science

https://colab.research.google.com/drive/1zP43cgA9w6zPutd2QQsww4K8IUj3OeN0?usp=sharing
Yaha baatcheet chal rahi hai.

---

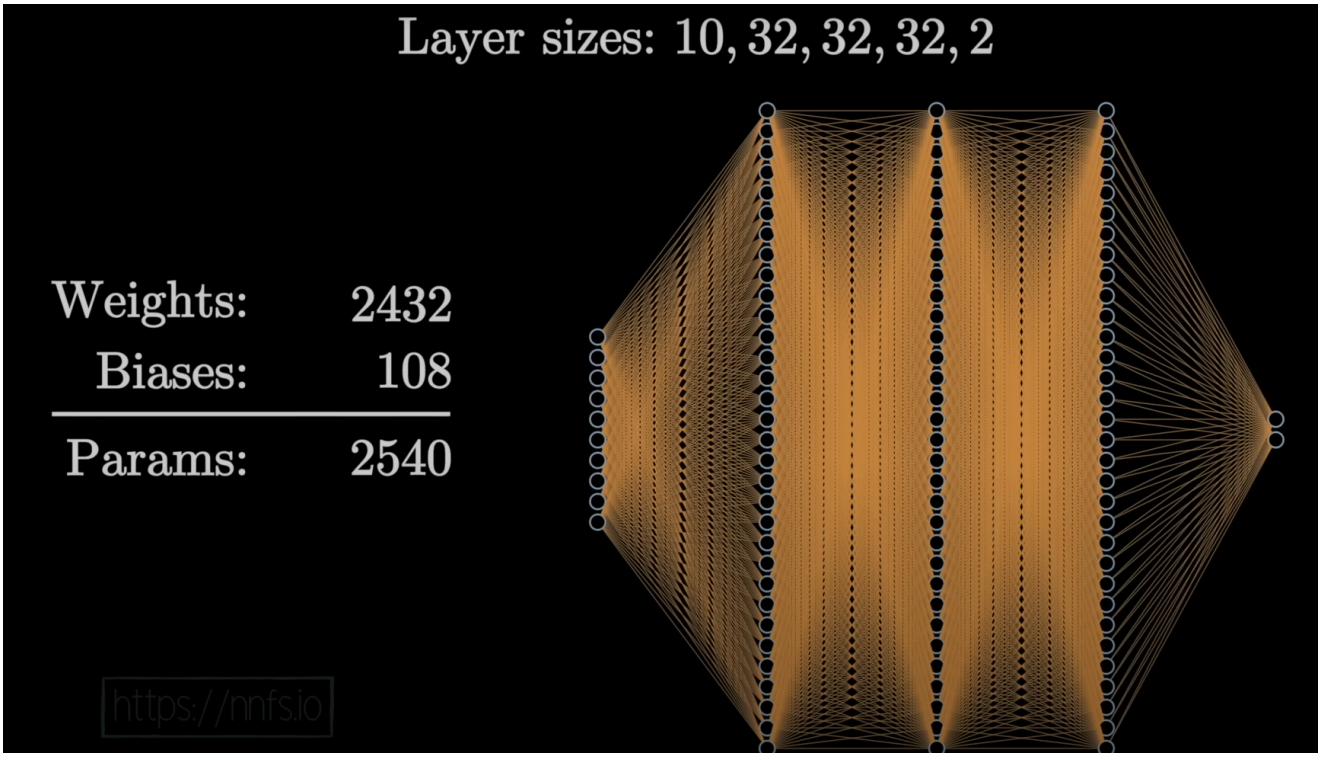
1. Neural Network:
   Make a simple neural neural network:
   `Step 1:`
   Machine learning consists of these steps:
   Gather **correctly labeled** data for the machine to train on.
   Create a metric to describe how much error the machine makes when trying to predict what something is.
   Iteratively train to reduce that error.

`Step 2`: Cost=number of errors/number of total predictions- A good model ha s a low cost.
Each neuron does apna kaam



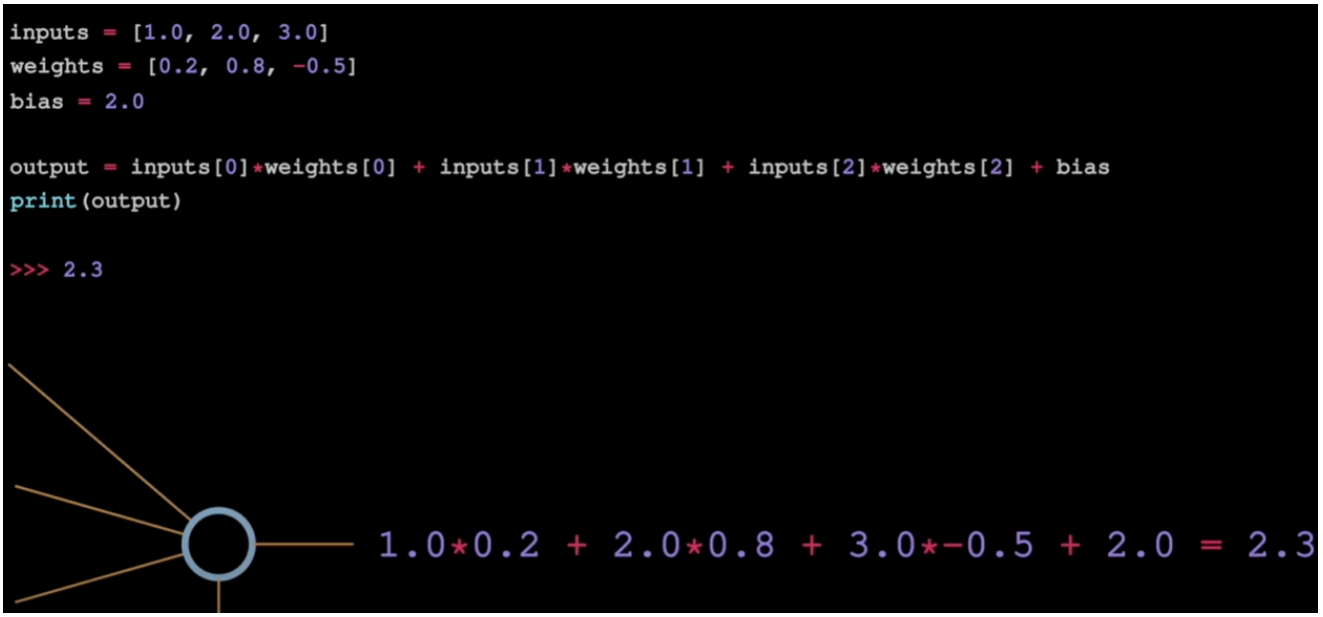
Weights and biases are tuned such that it can 'perdict' a value it can give

Every connection (orange line) is a unique weight and every neuron is unique bias

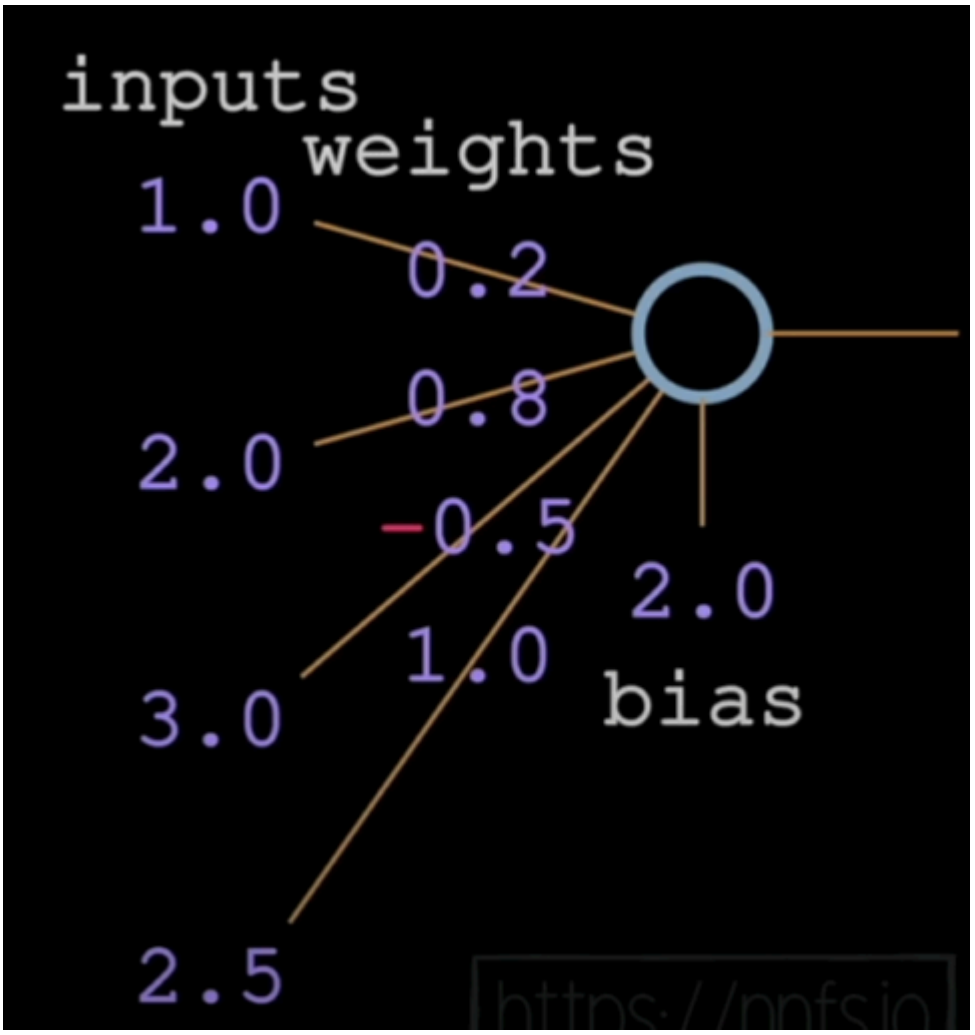


Hard Part- Tuning this.

For a simple program:



Corresponding image representation:



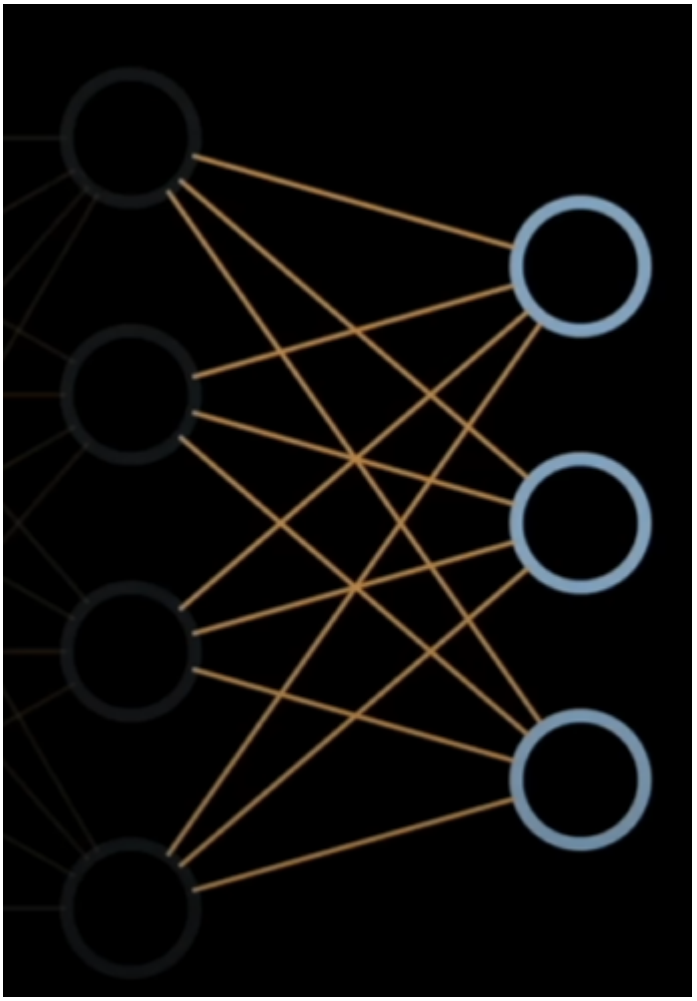
Above represents a 4 inputs, 4 weights modelling of a single neuron. The basic formula is what we stick to

$Output = Sigma(input[i] * weights[i] + bias)$

Now model 4 inputs into 3 neurons. This means 3 unique weight sets and biases.

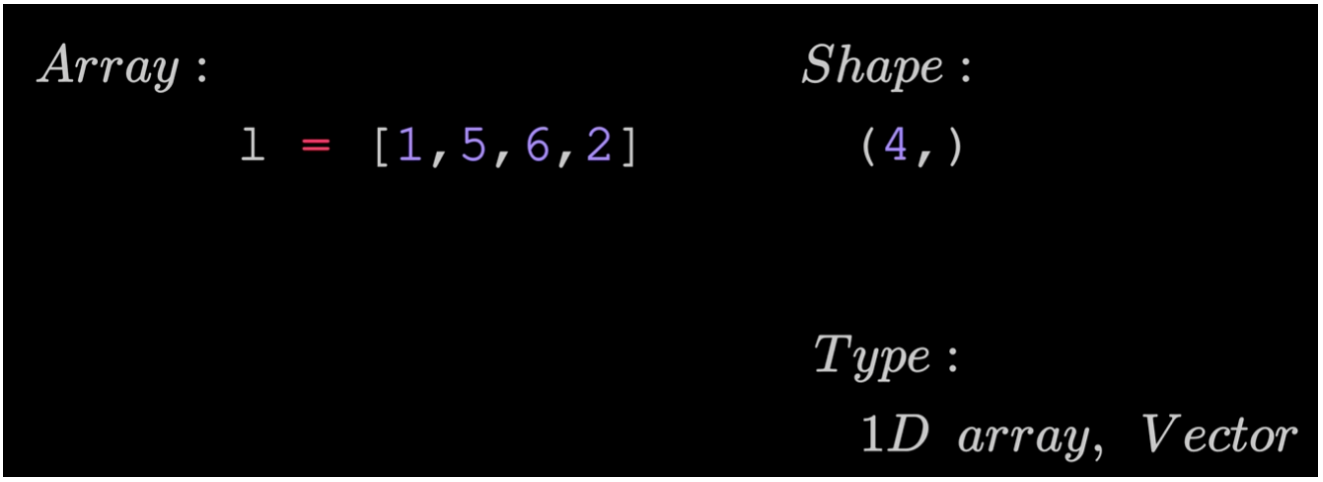
```
# Model say 3 nuerons with 4 inputs: So layer output would not be a single
value!!

inputs = [1,2,3,4]

weights = [0.2,0.8,-0.5,1.11]

weights1 = [0.3,0.5,-0.91,0.26]

weights2 = [0.44,0.3,-0.4,1.4]



bias=3

bias1=6.1
```

```
bias2=5

output= [inputs[0]*weights[0] + inputs[1]*weights[1] + inputs[2]*weights[2]
+ inputs[3]*weights[3] + bias, inputs[0]*weights1[0] + inputs[1]*weights1[1]
+ inputs[2]*weights1[2] + inputs[3]*weights1[3] + bias,
inputs[0]*weights2[0] + inputs[1]*weights2[1] + inputs[2]*weights2[2] +
inputs[3]*weights2[3] + bias]

print(output)
```

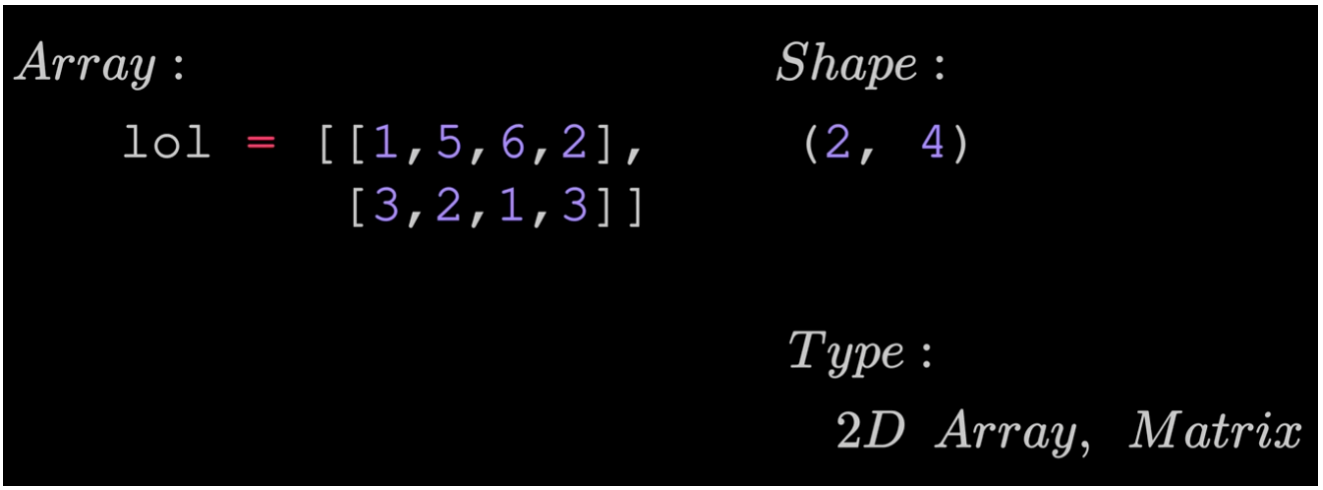


Now weights as a list of lists lete hai jab (same for biases)
We have concept of Activation programs- bias is offset or (intercept jaisa for $y = mx + c$) So 2 different tools (bias and weight) do 2 different works.

Concept of Shape:



1D Array- Numpy, Vecto- Mathematics



2D Array- Matric in Maths
The Arrays must be homologous- this means at each demention, size must be same.

Tensors- Object that can be represented as an array (not an array, but can represented it like that)

Dot Product ka concept= 2 vectors ke beech mein

```
#Now getting our hands dirty, we have:

#We have concept of Activation prograrms- bias is offset or (intercept jaisa for y=mx+c) So 2 different tools (bias and weight) do 2 different works.

inputs = [1,2,3,4]

weights = [[0.2,0.8,-0.5,1.11],[0.3,0.5,-0.91,0.26],[0.44,0.3,-0.4,1.4]]

biases=[3, 6.1, 5]
```

```
layer_outputs=[] #Output of current layer

for nw,nb in zip(weights,biases):

    output=0 #Output of given neuron

    for n1,w1 in zip(inputs,nw):

        output+=n1*w1

    output+=nb

    layer_outputs.append(output)

print(layer_outputs)
```

For a case of both Matrix and a Vector:
`import numpy as np  inputs = [1,2,3,4]`

```
weights = [[0.2,0.8,-0.5,1.11],[0.3,0.5,-0.91,0.26],[0.44,0.3,-0.4,1.4]]
```

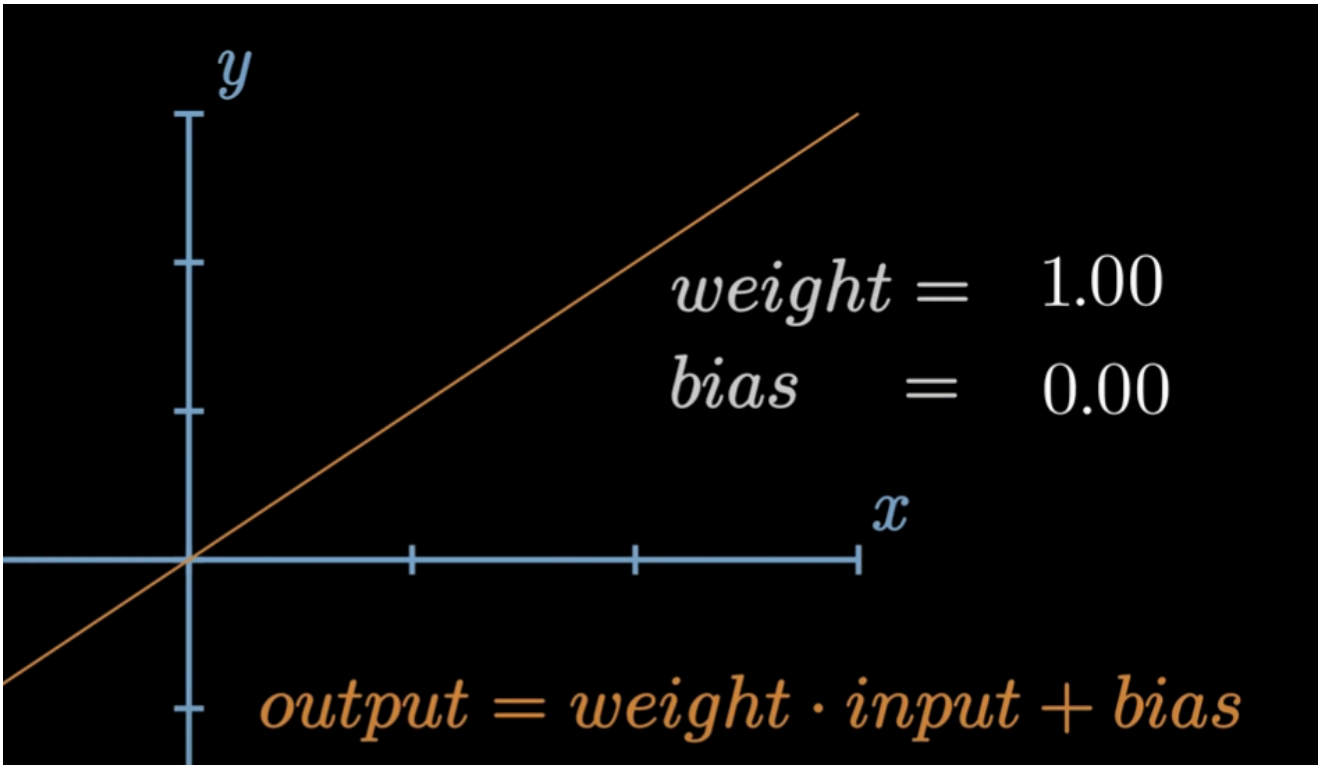
```
outputs= np.dot(weights, inputs)+ bias
```

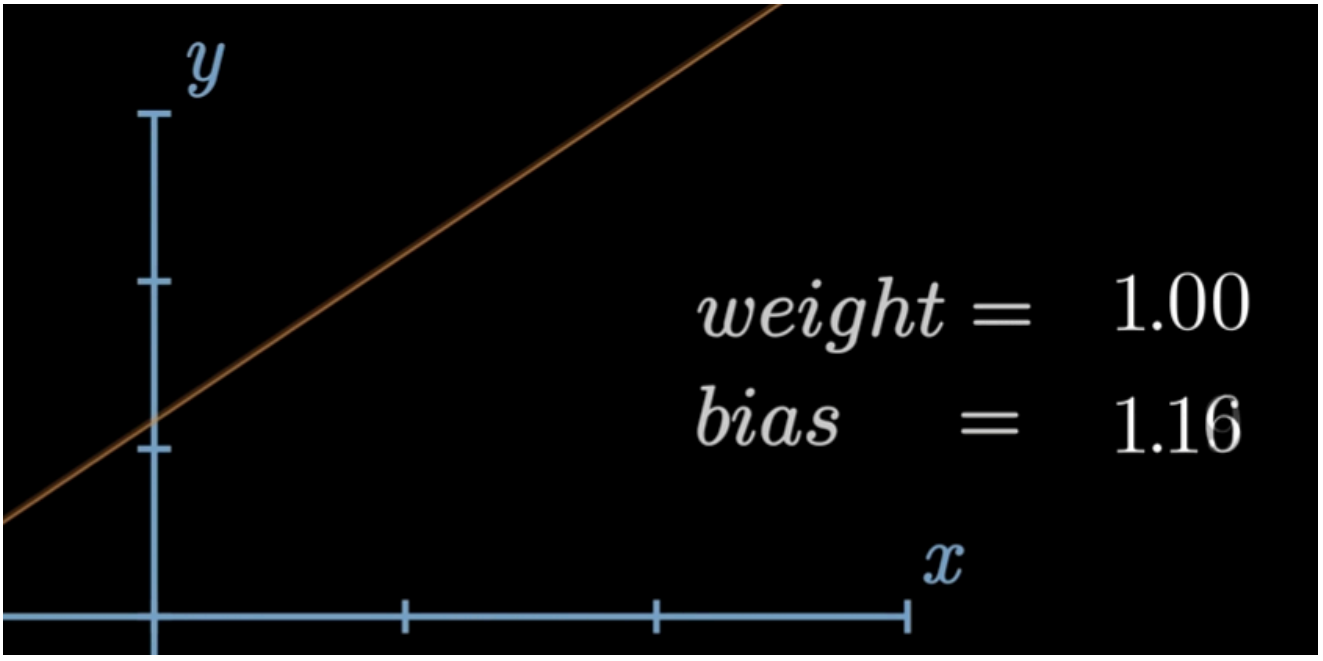
```
print(outputs)
```

Iska equivalent is as:

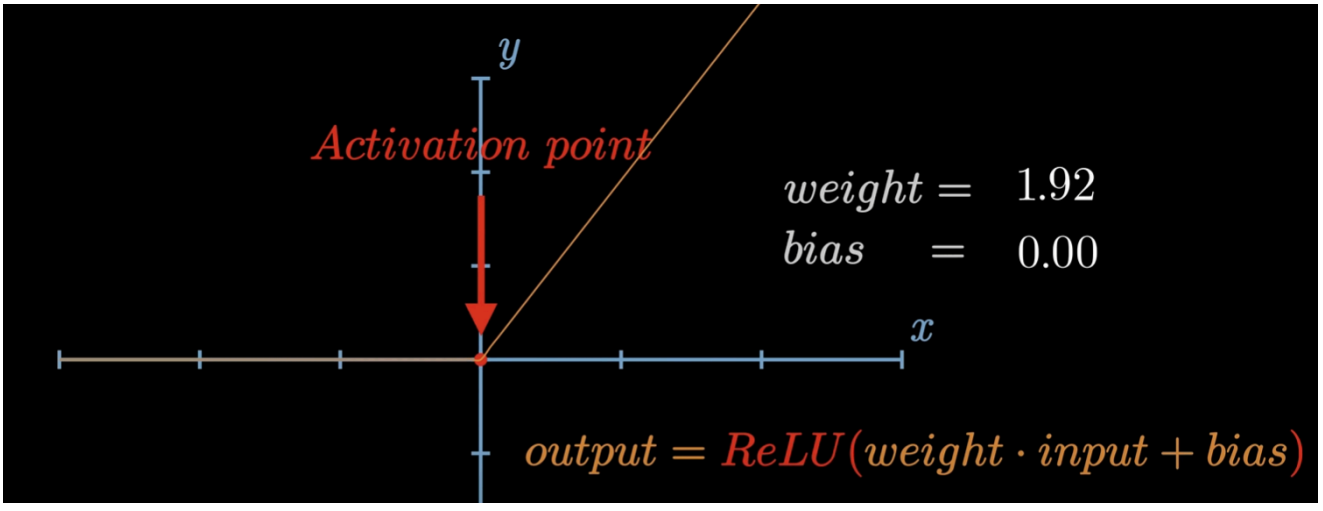
```
inputs = [1.0, 2.0, 3.0, 2.5]
weights = [[0.2, 0.8, -0.5, 1.0]
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
biases = [2.0, 3.0, 0.5]

outputs = np.dot(weights, inputs) + biases


np.dot(weights, inputs) = [np.dot(weights[0], inputs),
np.dot(weights[1], inputs), np.dot(weights[2], inputs)]
```

To say the least:



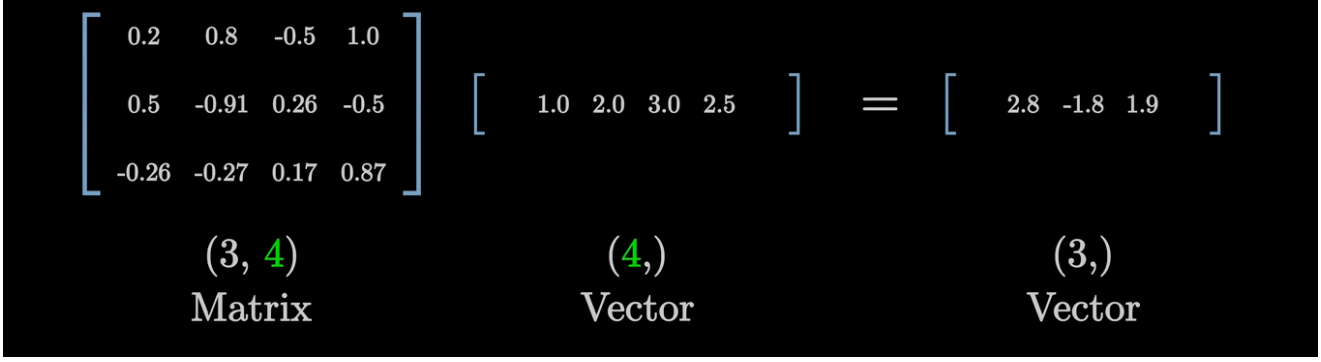


For activation functions;

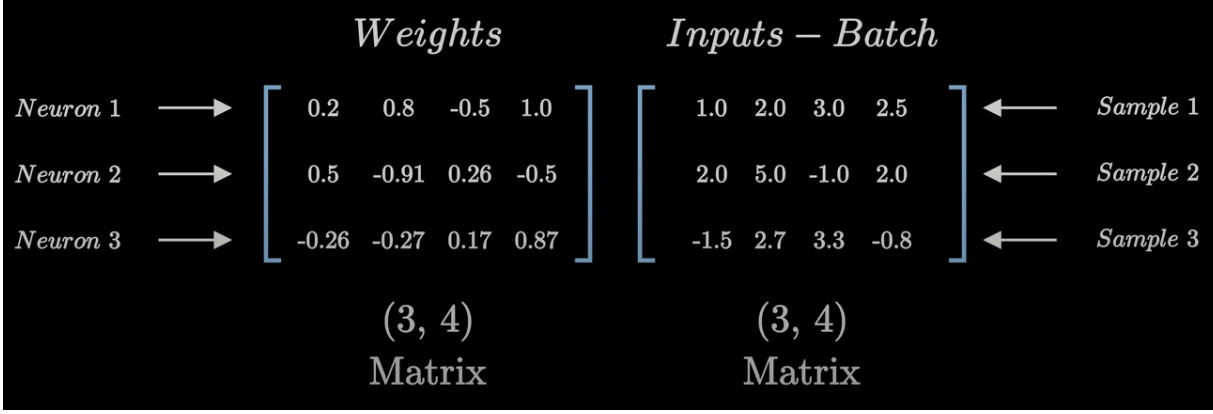


Errors ke liye refer to code- Shape Error

```
inputs = [1.0, 2.0, 3.0, 2.5]

weights = [[0.2, 0.8, -0.5, 1.0],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
```




```
inputs = [[1.0, 2.0, 3.0, 2.5],
          [2.0, 5.0, -1.0, 2.0],
          [-1.5, 2.7, 3.3, -0.8]]
weights = [[0.2, 0.8, -0.5, 1.0],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
```




```
outputs = np.dot(inputs, np.array(weights).T) + biases
```

```
inputs = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]

weights = [[0.2,0.8,-0.5,1.11],[0.3,0.5,-0.91,0.26],[0.44,0.3,-0.4,1.4]]

# Correcting the shape of weights2 to be compatible with layer1_outputs

weights2 = [[0.22,0.18,-0.25],[05.3,05.5,-03.91],[0.44,0.3,-0.4]] # Adjusted
```

```
weights2 to have 3 columns

biases=[3, 6.1, 5]

biases2=[3, 6, 0.9]

layer1_outputs= np.dot(inputs,np.array(weights).T)+ biases

layer2_outputs= np.dot(layer1_outputs,np.array(weights2).T)+ biases2 # Now
this multiplication is valid

print(layer2_outputs)
```

On adjusting weights;

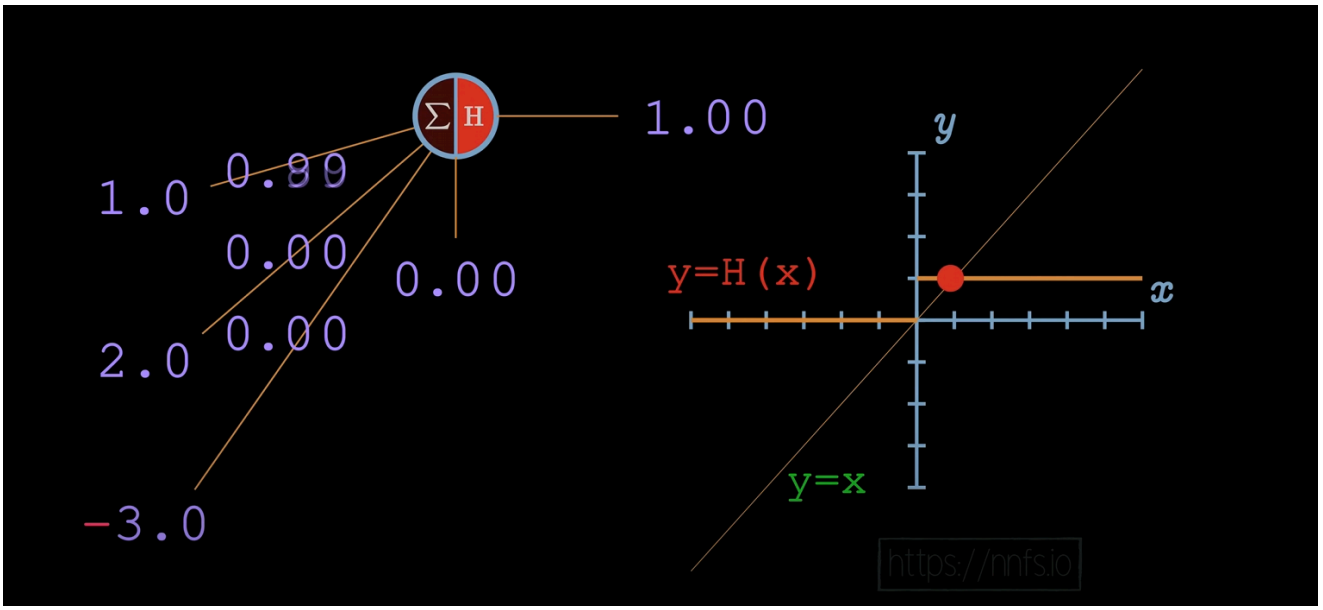
```
import numpy as np

X=[[1,2,3,2.5],[4,5,6,1.2],[7,8,9,1.45]]

np.random.seed(0)

class Layer_Dense: #We do that as Data passes through, it becomes bigger and
bigger till an explosion- Scaling of the dataset.

  def __init__(self, n_inputs, n_neurons):

    self.weights= np.random.randn(n_inputs ,n_neurons)

  def forward(self):

    pass

print(np.random.randn(4,3))
```

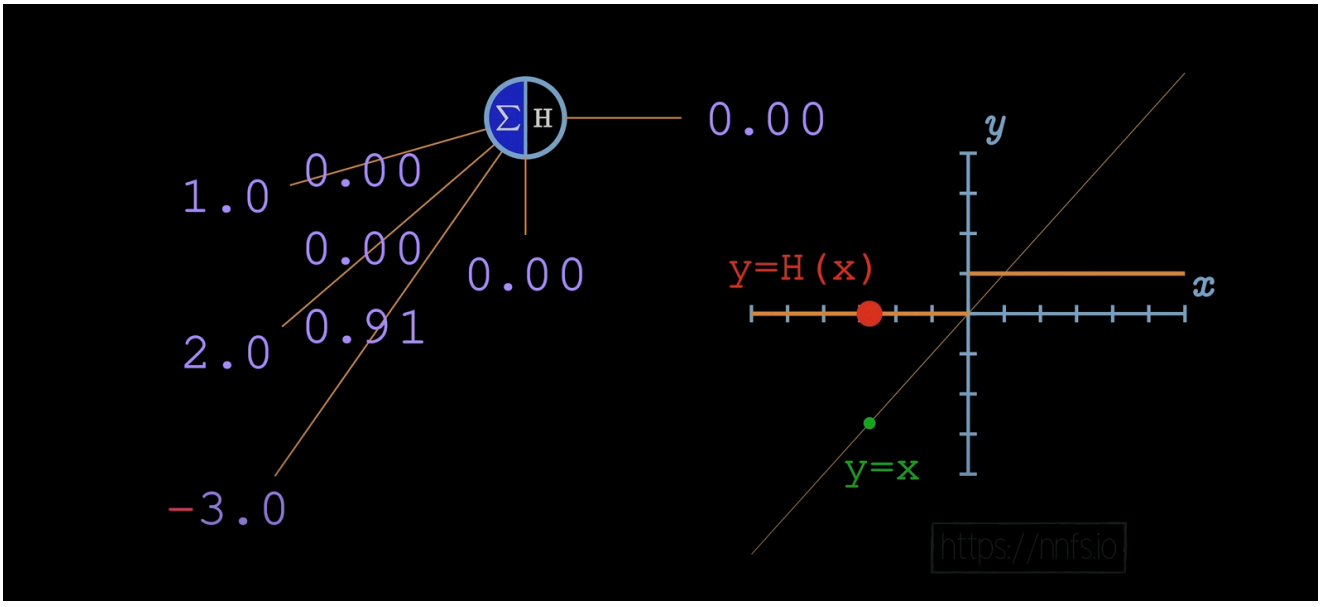
Forward method:

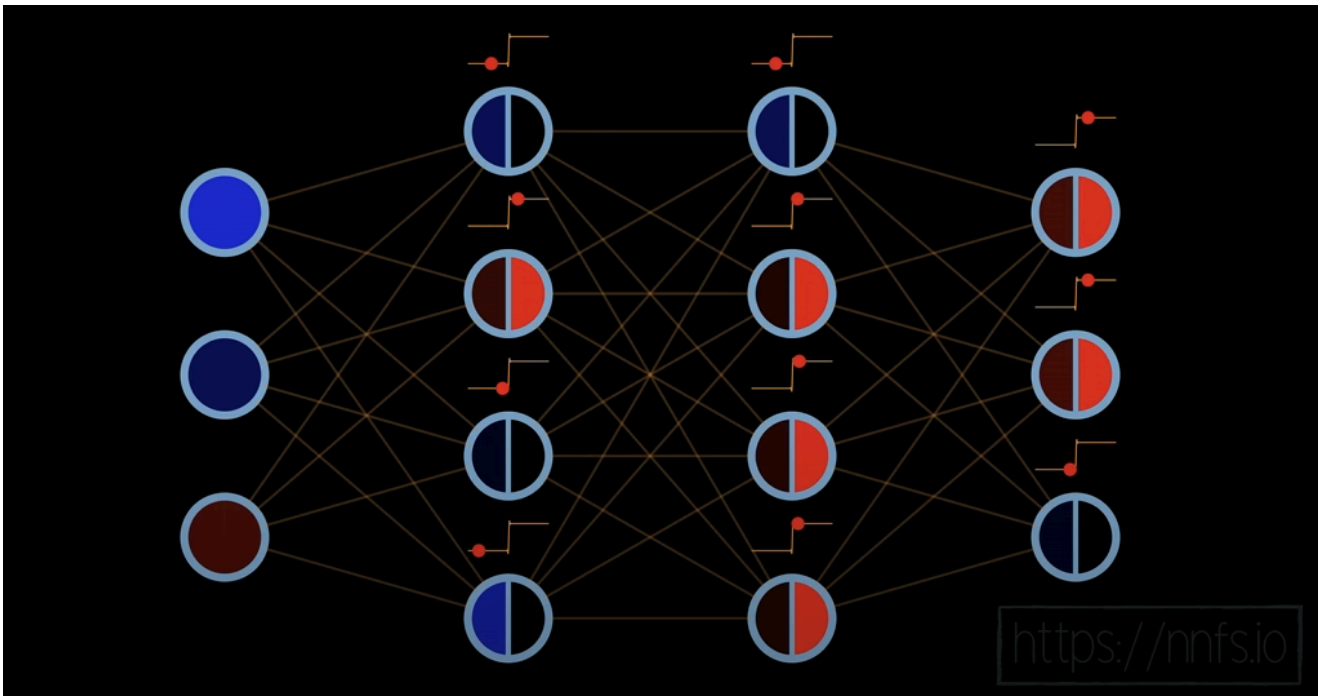
```
import numpy as np

X=[[1,2,3,2.5],[4,5,6,1.2],[7,8,9,1.45]]

np.random.seed(0)

class Layer_Dense: #We do that as Data passes through, it becomes bigger and
```

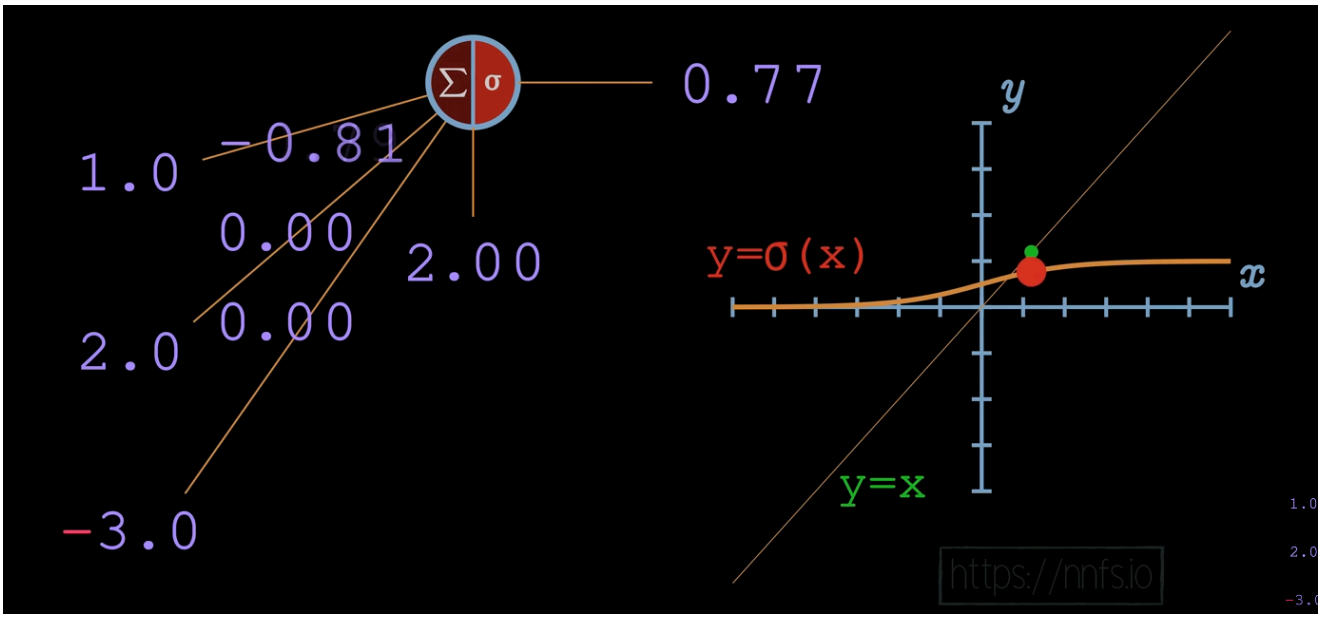
```
bigger till an explosion- Scaling of the dataset.

  def __init__(self, n_inputs, n_neurons):

    self.weights= 0.01* np.random.randn(n_inputs ,n_neurons)#Shaping the
weights

    self.biases=np.zeros((1,n_neurons))

  def forward(self, inputs): #Forward Method: takes the inputs

    self.output=np.dot(inputs,self.weights) +self.biases



layer_1= Layer_Dense(4,5)

layer_2= Layer_Dense(5,2)#5 yaha same hai, zaruri for Matrices

layer_1.forward(X)

print(layer_1.output)

layer_2.forward(layer_1.output)

print(layer_2.output)
```

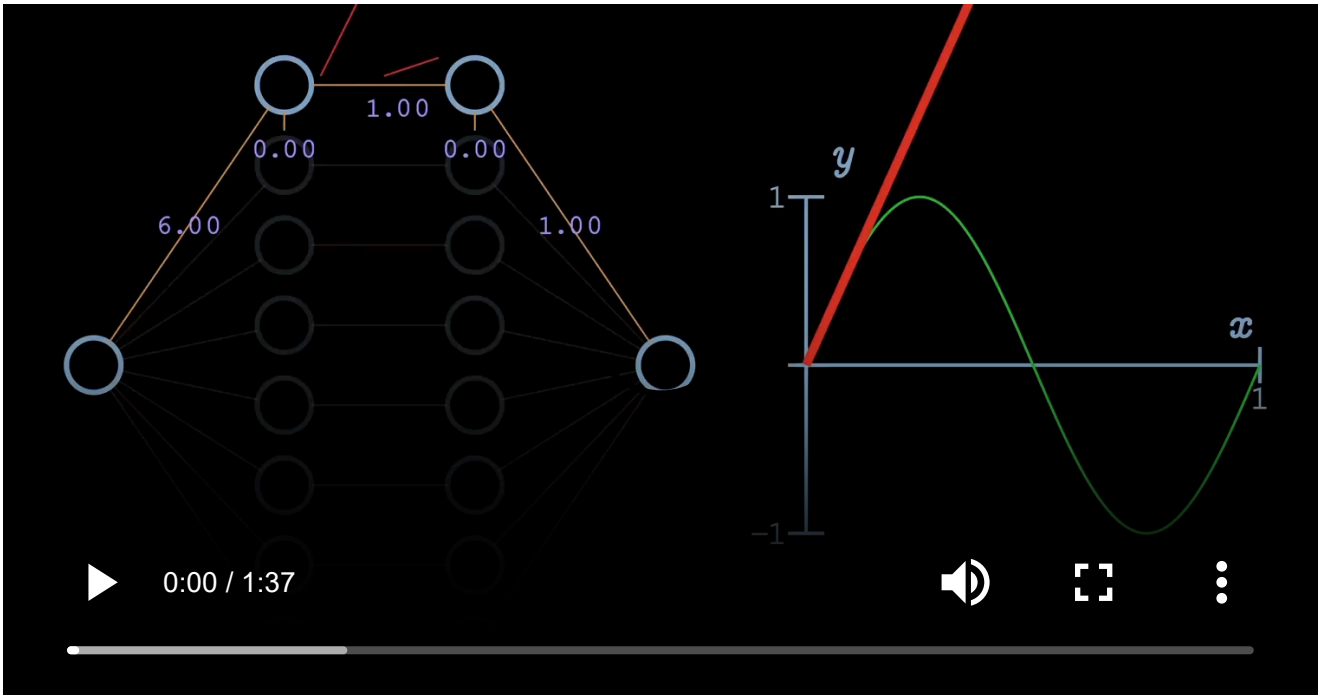
```
[[0.01075813 0.10398352 0.02446241 0.0318215  0.01885105]
 [0.03434491 0.16869607 0.07478202 0.0955321  0.11161489]
 [0.06310363 0.25656684 0.12192168 0.16409525 0.19114024]]
[[ 0.00148296 -0.00083976]
 [ 0.00403334 -0.00065392]
 [ 0.006379   -0.0007635 ]]
```

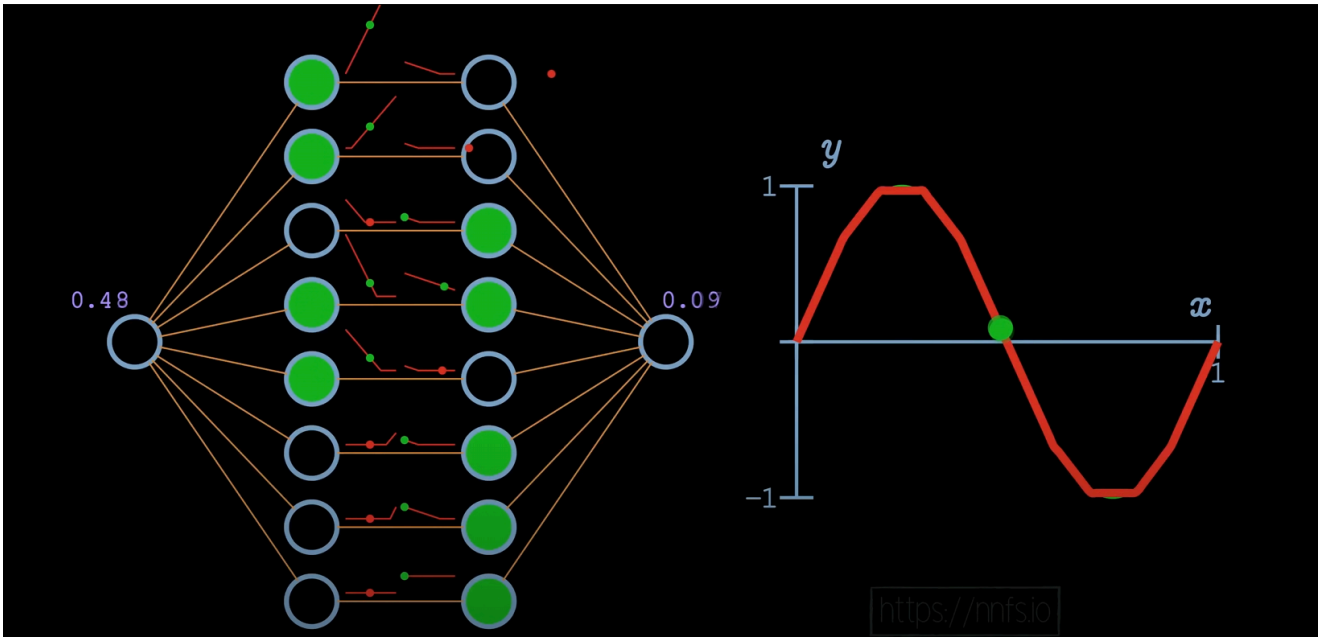





https://nnfs.io


Σ σ
0.77
1.0
-0.81
0.00
2.00
2.0
0.00
-3.0
y
y=σ(x)
x
y=x
https://nnfs.io


1.00
0.00
0.00
6.00
1.00
y
1
x
1
-1
0:00 / 1:37



Coding and running it without activation, we get:

```
import numpy as np
import nnfs
from nnfs.datasets import spiral_data
nnfs.init()

X=[[1,2,3,2.5],[4,5,6,1.2],[7,8,9,1.45]]
X,y = spiral_data(100,3)
class Layer_Dense:
  def __init__(self, n_inputs, n_neurons):
    self.weights= 0.01* np.random.randn(n_inputs ,n_neurons)
    self.biases=np.zeros((1,n_neurons))
  def forward(self, inputs):
    self.output=np.dot(inputs,self.weights) +self.biases


class Activation_ReLU:
  def forward(self,inputs):
    self.output=np.maximum(0,inputs)

layer_1= Layer_Dense(2,5) #No of inputs, No of neurons
activation_1= Activation_ReLU()
layer_1.forward(np.array(X))

print(layer_1.output)
```

```
[[ 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00
0.0000000e+00]
[-8.3581581e-05 -7.9040430e-05 -1.3345221e-04 4.6550449e-05
4.5684628e-06]
[-2.3999445e-04 5.9346880e-06 -2.2480826e-04 2.0357311e-05
6.1002436e-05]
...
[ 1.1329151e-02 -1.8926226e-02 -2.0685506e-03 8.1107961e-03
-6.7135082e-03]
[ 1.3458835e-02 -1.4319782e-02 3.0949395e-03 5.6633754e-03
-6.2968740e-03]
[ 1.0781791e-02 -2.0080963e-02 -3.3757931e-03 8.7256189e-03
-6.8145879e-03]]
```

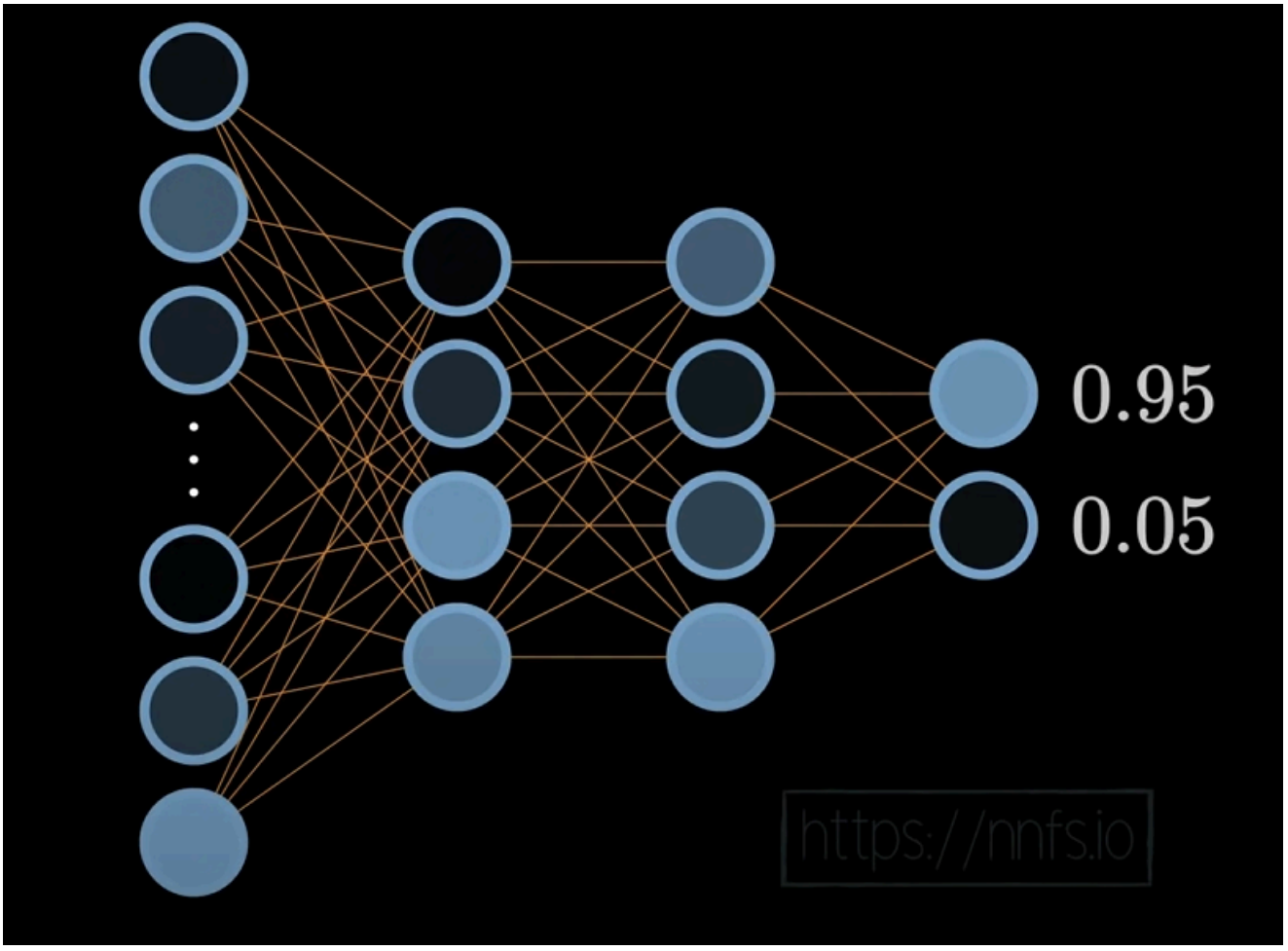
We get the above Output, we see many negative values.

On running with the activation function now,

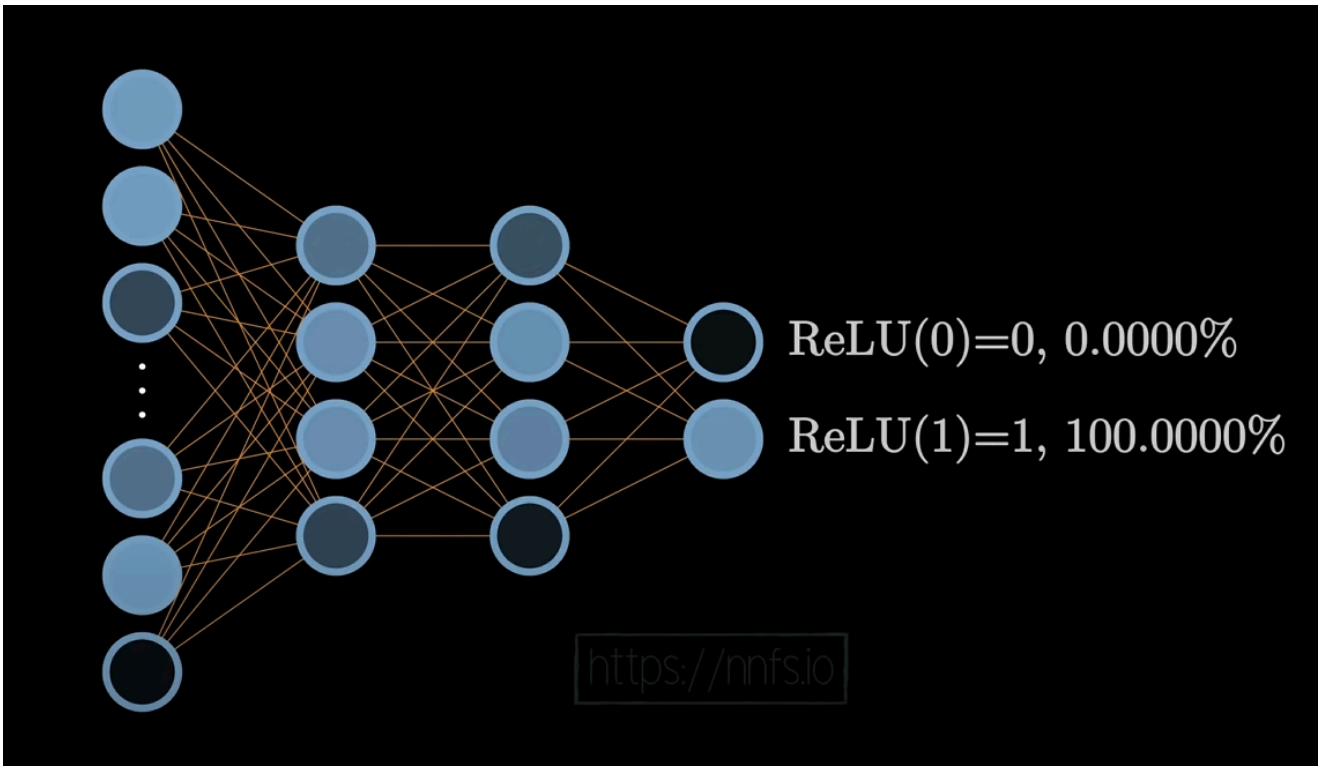
```
import numpy as np
import nnfs
from nnfs.datasets import spiral_data

nnfs.init()
X=[[1,2,3,2.5],[4,5,6,1.2],[7,8,9,1.45]]

X,y = spiral_data(100,3)
class Layer_Dense:
  def __init__(self, n_inputs, n_neurons):
    self.weights= 0.01* np.random.randn(n_inputs ,n_neurons)
    self.biases=np.zeros((1,n_neurons))
  def forward(self, inputs):
    self.output=np.dot(inputs,self.weights) +self.biases

class Activation_ReLU:
  def forward(self,inputs):
    self.output=np.maximum(0,inputs)
layer_1= Layer_Dense(2,5) #No of inputs, No of neurons
activation_1= Activation_ReLU()
layer_1.forward(np.array(X))
activation_1.forward(layer_1.output)
print(activation_1.output)
```

```
[[0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00]
 [0.0000000e+00 0.0000000e+00 0.0000000e+00 4.6550449e-05 4.5684628e-06]
 [0.0000000e+00 5.9346880e-06 0.0000000e+00 2.0357311e-05 6.1002436e-05]
 ...
 [1.1329151e-02 0.0000000e+00 0.0000000e+00 8.1107961e-03 0.0000000e+00]
 [1.3458835e-02 0.0000000e+00 3.0949395e-03 5.6633754e-03 0.0000000e+00]
 [1.0781791e-02 0.0000000e+00 0.0000000e+00 8.7256189e-03 0.0000000e+00]]
```

Softmax Activation Function: Used specifically for Output layer

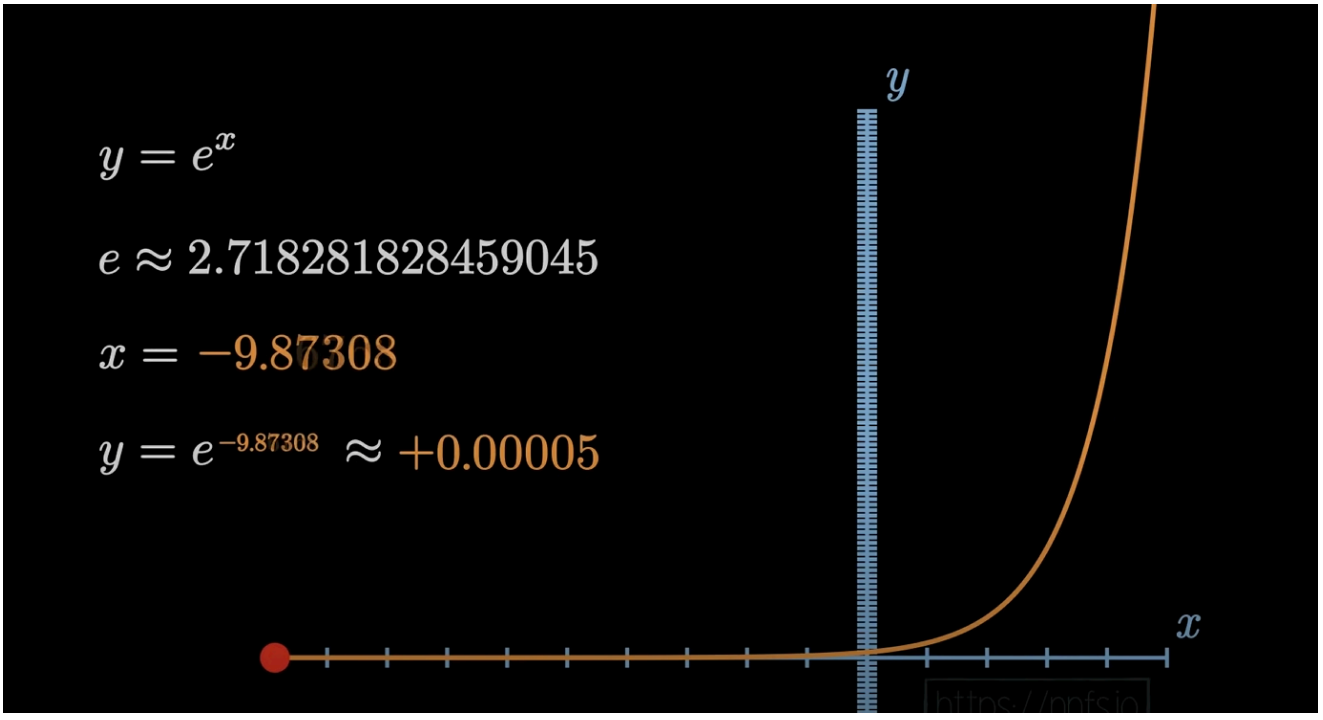


For an input, suppose cat or not, we get this as the output- probabilities. As an immediate idea:



We here are now going to solve the negative values, which exist even with a Linear Rectification function.

A solution could be:



Handles the Negative Values well.

Normalization ka concept suggests that:

$$y = \frac{u}{\sum_{i=1}^{n} u_i}$$

$$u = \begin{bmatrix} 1 \leftarrow u_1 \\ 2 \leftarrow u_2 \end{bmatrix}$$

$$y = \begin{bmatrix} \frac{u_1}{u_1+u_2} \\ \frac{u_2}{u_1+u_2} \end{bmatrix} = \begin{bmatrix} \frac{1}{1+2} \\ \frac{2}{1+2} \end{bmatrix} = \begin{bmatrix} \frac{1}{3} \\ \frac{2}{3} \end{bmatrix} = \begin{bmatrix} 0.33 \\ 0.67 \end{bmatrix}$$

But we can still have negative values in the above case, so we use exp.

So what are we doing?

$$\textit{Input} \rightarrow \textit{Exponentiate} \rightarrow \textit{Normalize} \rightarrow \textit{Output}$$

$$\begin{matrix} \textit{Dog} \rightarrow \\ \textit{Cat} \rightarrow \\ \textit{Human} \rightarrow \end{matrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \rightarrow \begin{bmatrix} e^1 \\ e^2 \\ e^3 \end{bmatrix} \rightarrow \begin{bmatrix} \frac{e^1}{e^1+e^2+e^3} \\ \frac{e^2}{e^1+e^2+e^3} \\ \frac{e^3}{e^1+e^2+e^3} \end{bmatrix} \rightarrow \begin{bmatrix} 0.09 \\ 0.24 \\ 0.67 \end{bmatrix}$$

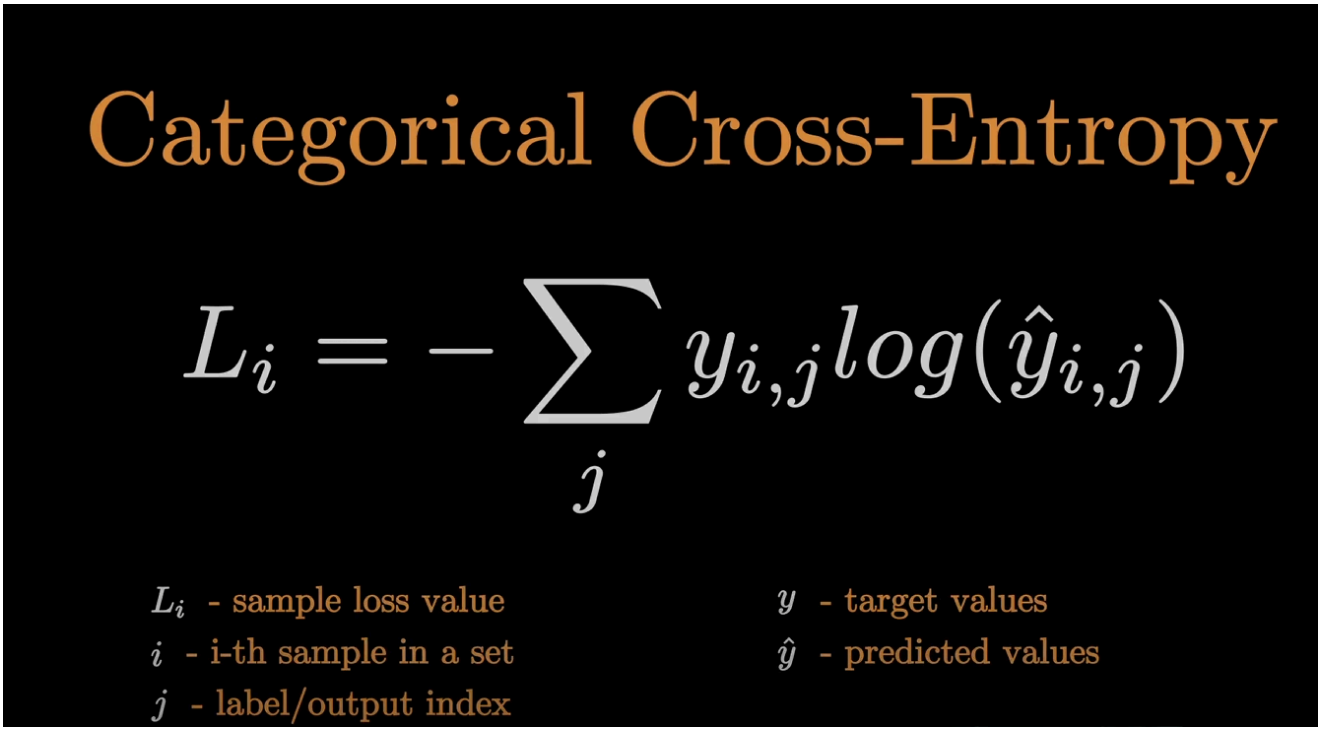
This is the Softmax Activation Function.

Code implementation of a working 2 layered Model:

```
import math
import numpy as np
import nnfs
```

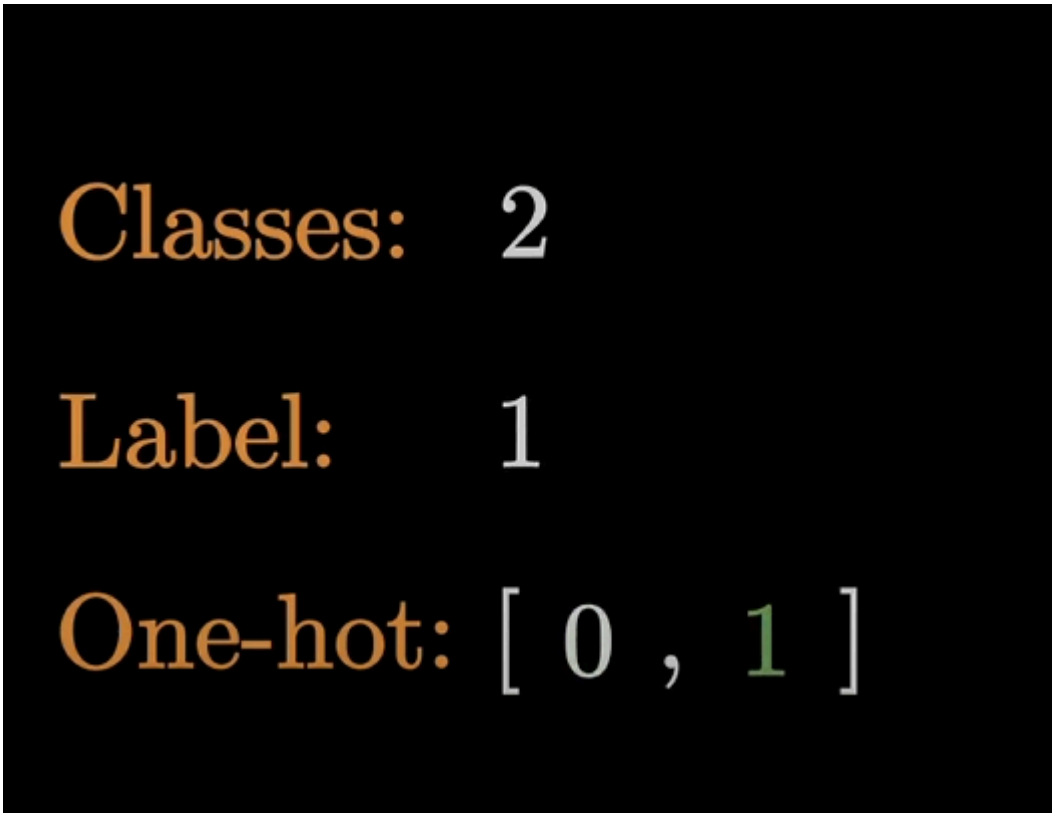
```
nnfs.init()
class Layer_Dense:
  def __init__(self,n_inputs,n_neurons):
    self.weights=0.1*np.random.randn(n_inputs,n_neurons)
    self.biases=np.zeros((1,n_neurons))
  def forward(self,inputs):
    self.output=np.dot(inputs,self.weights) + self.biases
class Activation_ReLU:
  def forward(self,inputs):
    self.output=np.maximum(0,inputs)
class Activation_Softmax:
  def forward (self,inputs):
    exp_values =np.exp(inputs- np.max(inputs, axis=1,keepdims=True))
    probabilities= exp_values/np.sum(exp_values, axis=1, keepdims=True)
    self.output=probabilities
X,y = spiral_data(samples=100, classes=3)
dense1= Layer_Dense(2,3)
activation1= Activation_ReLU()

dense2=Layer_Dense(3,3)
activation2= Activation_Softmax()
dense1.forward(X)
activation1.forward(dense1.output)
dense2.forward(activation1.output)
activation2.forward(dense2.output)
print(activation2.output[:5])


[[0.33333334 0.33333334 0.33333334] [0.33331734 0.3333183 0.33336434]
[0.3332888 0.33329153 0.33341965] [0.33325943 0.33326396 0.33347666]
[0.33323312 0.33323926 0.33352762]]
```

Loss/Error: Use **Categorical Cross Entropy**

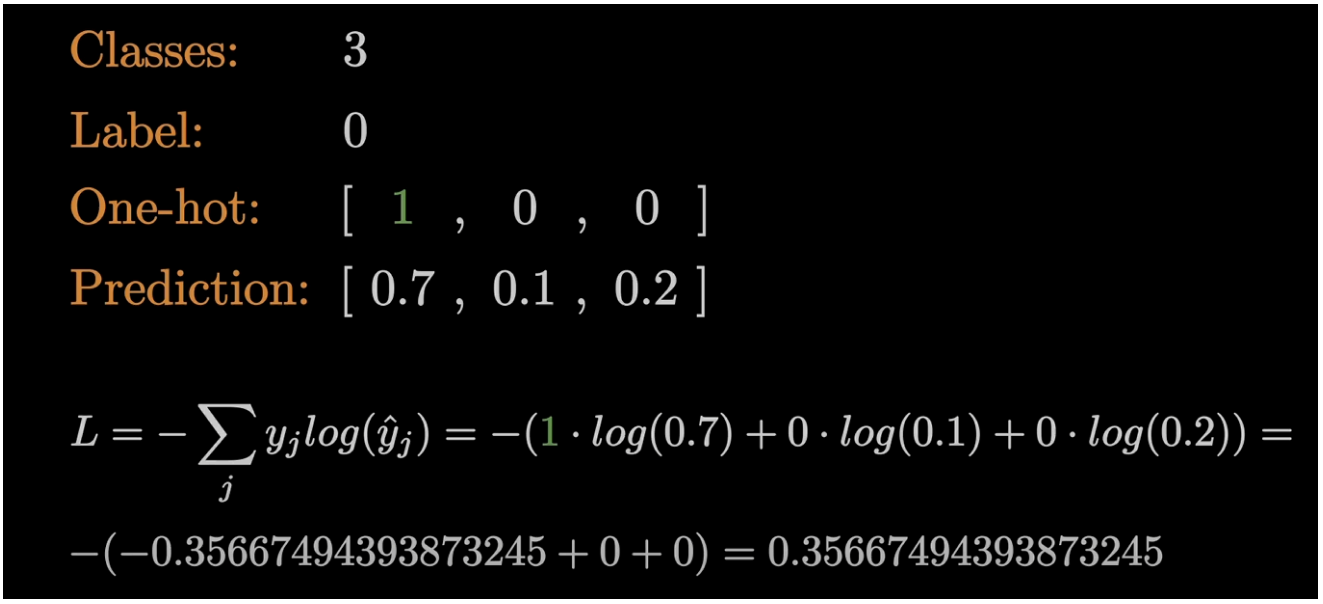


One of the many techniques- successful an convinient.

One Hot Encoding:
A vector of n classes, has 1 as label , so 1th one hot will have a 1



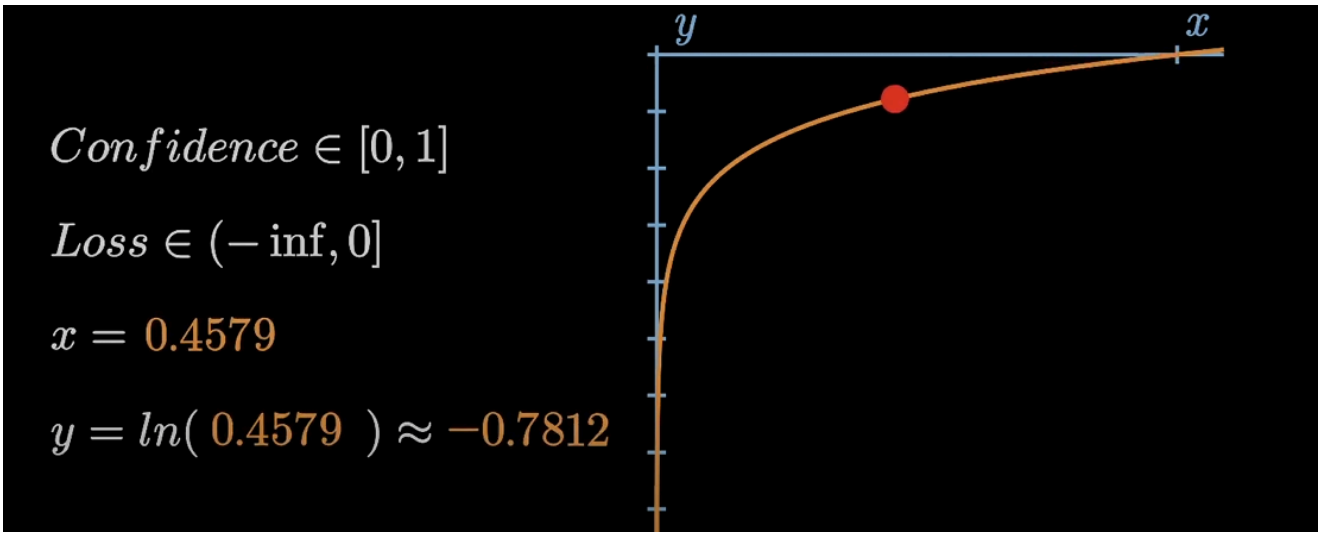
How it is implemented:



Code on how error can be found:
`import math

```
softmax_output=[0.7,0.1,0.2]
target_output= [1,0,0]
loss =-
math.log(softmax_output[0])*target_output[0]+math.log(softmax_output[1])*target_output[1]+math.log(softmax_output[2])*target_output[2])
print(loss)
```

same as using only target ka 0th element



The trend that the error show-

Implementing Loss:

```
softmax_outputs = [[0.7,  0.1,  0.2 ],
                   [0.1,  0.5,  0.4 ],
                   [0.02, 0.9,  0.08]]


Classes :
  0 – dog
  1 – cat
  2 – human

class_targets = [ 0 ,  1 ,  1 ]
```

Confidences on correct labels:

0.7

0.5

0.9 (har row se class-th element pick karo)

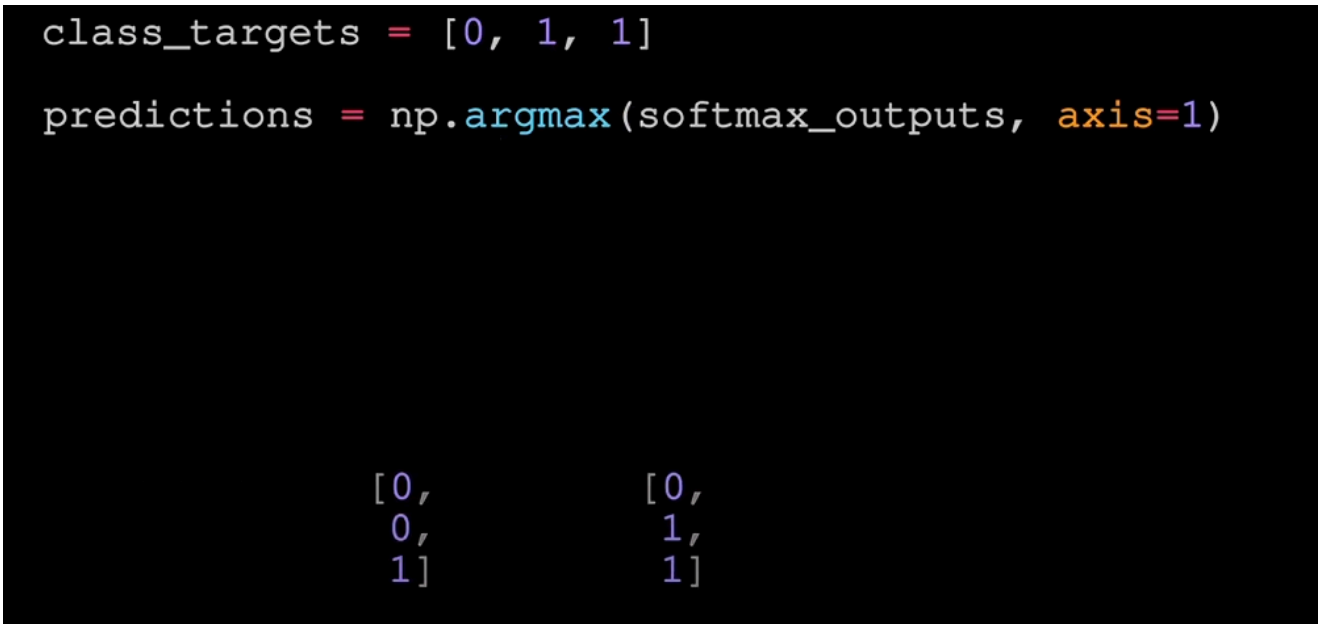
Use numpy array for it to be faster.

```
if len(y_true.shape) == 1:
    correct_confidences = y_pred_clipped[range(samples), y_true]


[[0.7,  0.1,  0.2 ],     [  [0,     [0,  ]  =    [0.7,
 [0.1,  0.5,  0.4 ],      [  1,      1,  ]        0.5,
 [0.02, 0.9,  0.08]]      [  2]  ,   1]  ]        0.9]
```

How we select to keep loss for y shape =1 (One D array)

for 2D array:

```
if len(y_true.shape) == 2:
    correct_confidences = np.sum(y_pred_clipped*y_true), axis=1)


[[0.7,  0.1,  0.2 ],        [[1, 0, 0],        [[0.7, 0.0, 0.0],
 [0.1,  0.5,  0.4 ],   *     [0, 1, 0],   =     [0.0, 0.5, 0.0],
 [0.02, 0.9,  0.08]]         [0, 1, 0]]         [0.0, 0.9, 0.0]]
```



To find accuracy, in the batch of outputs- axis one of the softmax and class_targets compare them side by side- if both same then resuting 1 else 0
Accuracy is iska average basically.

Accuracy is useful, but loss metric is metric of how wrong something is.

Optimized Code:

```
import numpy as np
import nnfs
from nnfs.datasets import spiral_data

# Initialize nnfs for consistent data generation and float32 precision
nnfs.init()

# Create dataset: 100 samples per class, 3 classes
X, y = spiral_data(samples=100, classes=3)

# Fully connected (dense) layer
class Layer_Dense:
    def __init__(self, n_inputs, n_neurons):
        # Initialize weights with small random values and biases with zeros
        self.weights = 0.1 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))

    def forward(self, inputs):
        self.output = np.dot(inputs, self.weights) + self.biases

# ReLU Activation Function
class Activation_ReLU:
    def forward(self, inputs):
```

```
        self.output = np.maximum(0, inputs)

# Softmax Activation Function (for output layer)
class Activation_Softmax:
    def forward(self, inputs):
        # Subtract max for numerical stability
        exp_values = np.exp(inputs - np.max(inputs, axis=1, keepdims=True))
        # Normalize to probabilities
        probabilities = exp_values / np.sum(exp_values, axis=1,
keepdims=True)
        self.output = probabilities

# Categorical Cross-Entropy Loss
class Loss_CategoricalCrossentropy:
    def forward(self, y_pred, y_true):
        # Number of samples
        samples = len(y_pred)

        # Clip to prevent log(0)
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Probabilities for correct classes
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[range(samples), y_true]
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(y_pred_clipped * y_true, axis=1)

        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return np.mean(negative_log_likelihoods)

# Build the network
dense1 = Layer_Dense(2, 3)          # 2 inputs → 3 neurons
activation1 = Activation_ReLU()     # ReLU after first layer

dense2 = Layer_Dense(3, 3)          # 3 inputs (from dense1) → 3 output
neurons
activation2 = Activation_Softmax()  # Softmax for output



dense1.forward(X)
activation1.forward(dense1.output)
```

```
dense2.forward(activation1.output)
activation2.forward(dense2.output)

#loss
loss_function = Loss_CategoricalCrossentropy()
loss = loss_function.forward(activation2.output, y)

print("First 5 outputs (after Softmax):\n", activation2.output[:5])
print("\nLoss:", loss)
```