# Neural Network from Scratch:-

input = [1, 2, 3]
weights = [0.2, 0.8, -0.5]
bias = 2

Single Neuron (Dot product)

O/p:- input[0] * weight[0] + input[1] * weight[1] + input[2] * weight[2] + bias

Layer:- n neurons

inputs:- [1, 2, 3, 2.5]

1 neurons { n inputs, 1 bias }

weights1 = [0.2, 0.8, -0.5, 1]
weights2 = [0.5, -0.91, 0.26, -0.5]
weights3 = [-0.26, -0.27, 0.17, 0.87]
bias1 = 2
bias2 = 3
bias3 = 0.5

O/p Layer:- ( inputs * weights1 + bias1, inputs * weight2 + bias2, inputs * weights3 + bias3 ]

Shape:- Dimensions

array:- homogeneous

Eg:- [[[1, 5, 6, 2], [3, 2, 1, 3]], [[5, 2, 1, 2], [6, 4, 8, 4]], [[2, 8, 5, 3], [1, 1, 9, 4]]]

Shape ⇒ (3, 2, 4)

total ⇒

Vector = list (1-D Array)

tensor object → object that can be represented as an array

input related not neuron related.

```
layer_output = []

for n_weight, n_bias in zip(weights, biases):

    n_op = 0
    for n_input, weight in zip(input, n_weight):

        n_op += n_input * weight.

    n_op += n_bias

    layer_output.append(n_op)
```

⇓ Numpy (np)

```
layer_output = np.dot(weight, inputs) + biases
```

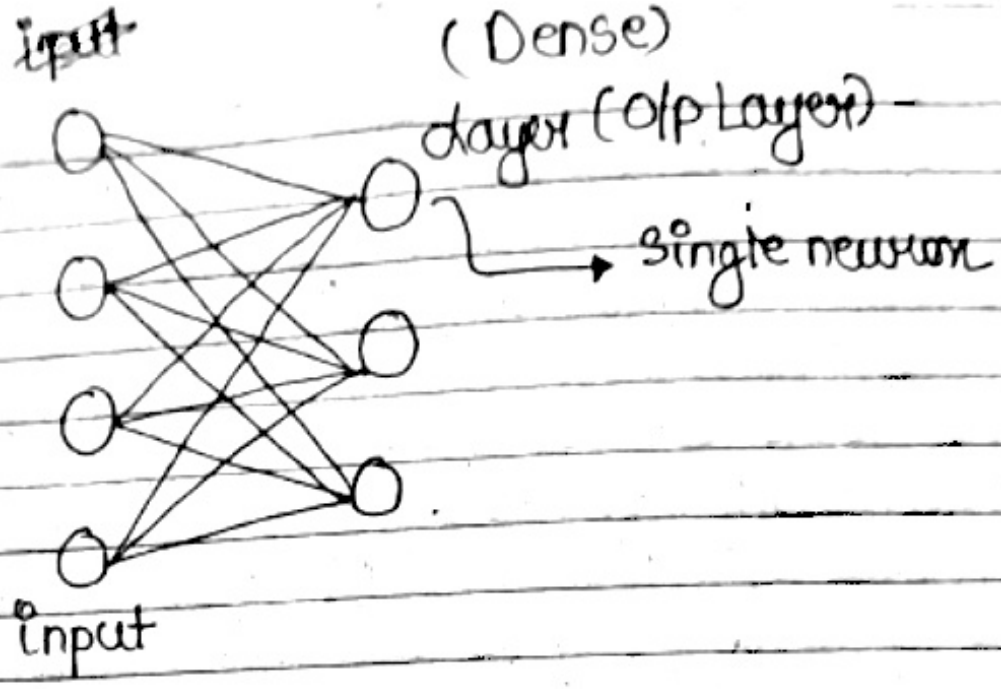
~~(1-D) - out~~

⇓ Batch of inputs (2D)

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

fig:-

```
inputs = [[1, 2, 3, 2.5]
          [2, 5, -1, 2]
          [-1.5, 2.7, 3.3, -0.8]]
weights = [[0.2, 0.8, -0.5, 1],
           [0.5, -0.91, 0.26, -0.5]
           [-0.26, -0.27, 0.17, 0.87]]
biases = [2.0, 3.0, 0.5]
```




fg Dataset :- spiral-data from nnfs

dots-feature
Coordinate :- samples
class :- 0, 1, 2

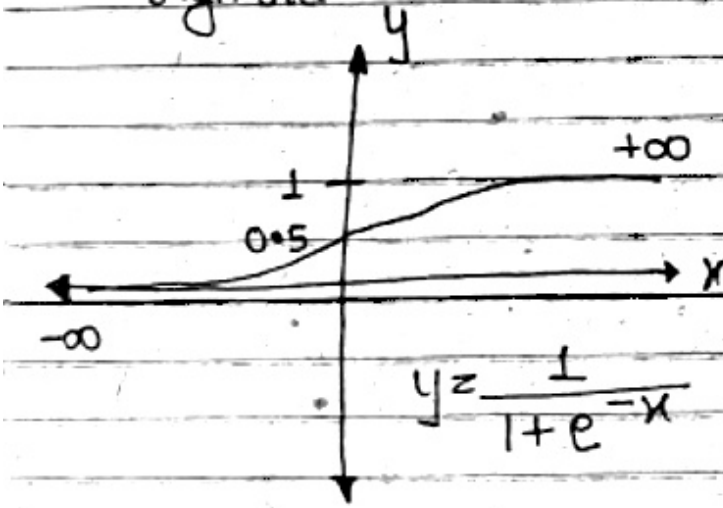
Class Layer_Dense:-

```
def __init__(self, n_inputs, n_neurons):
    self.weights = 0.01 * np.random.
                   randn(n_inputs, n_neurons)
    self.biases = np.zeros((1, n_neuron))

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

Activation functions:-

Sigmoid

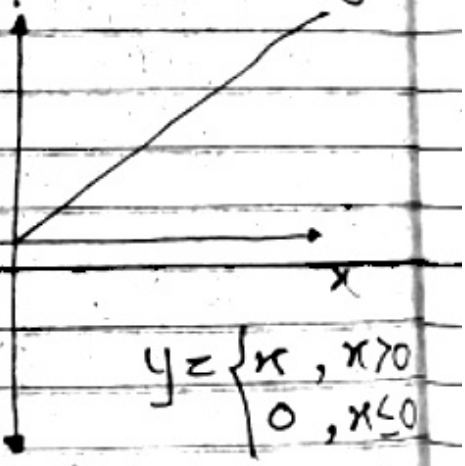



$$y = \frac{1}{1+e^{-x}}$$

computationally heavy

ReLU

Speed, efficiency




$$y = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

class Activation_ReLU:

```
def forward(self, input):
    self.output =
    np.maximum(0, input)
```

axis=0 → rows, axis=1 → columns

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
```

Categorical Cross-Entropy Loss:-

$L_i = -\log(y_{i,k})$

class Loss:-

```
def calculate(self, output, y)
    sample_loss = self.forward(output, y)
    data_loss = np.mean(sample_losses)
    return data_loss.
```

```
Class Loss - CategoricalCrossEntropy (Loss):
    def forward (self, y_pred, y_true):
        samples = len(y_pred)
        y_pred_clipped =
            np.clip(y_pred, 1e-7,
                    1-1e-7)
        if len(y_true.shape)==1:
            correct_confidence = y_pred_clipped
                [range(samples), y_true]
        elif len(y_true.shape)==2:
            correct_confidence = np.sum(
                y_pred_clipped * y_true,
                axis=1)
        negative_log_likelihood = -np.log
                (correct_confidences)
        return negative_log_likelihood.

loss_function = Loss_CategoricalCrossEntropy()
loss = loss_function.calculate(softmax_outputs,
                               class_targets)
print(loss)
```

Accuracy:-

```
predictions = np.argmax(softmax_outputs, axis=1)
if len(class_target.shape) == 2:
    class_targets = np.argmax(class_targets,
                              axis=1)
accuracy = np.mean(predictions == class_targets)
```

Gradients

The gradient is a vector composed of all of the partial derivatives of a function, calculated with each input variable.

$$\frac{\partial}{\partial x_0}[ReLU(sum(mul(x_0,w_0), mul(x_1,w_1), mul(x_2,w_2), b))]$$

$$= \frac{\partial ReLU()}{\partial sum()} * \frac{\partial sum()}{\partial mul(x_0,w_0)} * \frac{\partial mul(x_0,w_0)}{\partial x_0}$$

relu_dz = (1 if z > 0 else 0)

drelu_dz = dvalue * (1. if z > 0 else 0)

dvalue ↓ received from next layer (propagated Backward)

```
Class Layer_Dense:
    ...
    def backward(self, dvalues):
        self.dweights = np.dot(self.inputs.T,
                               dvalues)
        self.dbiases = np.sum(dvalues, axis=0
                              , keepdims=True)
        self.dinputs = np.dot(dvalues,
                              self.weights.T)
```

Activation

```
Class ~~RELU~~ - ReLU:
    def forward(self, inputs):
        self.inputs = inputs
        self.output = np.maximum(0, inputs)
    def backward(self, dvalues):
        self.dinputs = dvalues.copy()
        self.dinputs[self.inputs <= 0] = 0
```

```
class Loss_CategoricalCrossEntropy(Loss):
    ...
    def backward(self, dvalues, y_true):
        samples = len(dvalues)
        labels = len(dvalues[0])
        if len(y_true.shape) == 1:
            y_true = np.eye(labels)[y_true]
        self.dinputs = -y_true / dvalues
        self.dinputs = self.dinputs / samples

class Activation_Softmax:
    ...
    def backward(self, dvalues):
        self.dinputs = np.empty_like(dvalues)
        for index, (single_output, single_dvalues) in
        enumerate(zip(self.output, dvalues)):
            single_output = single_output.
                            reshape(-1, 1)
            jacobian =
```