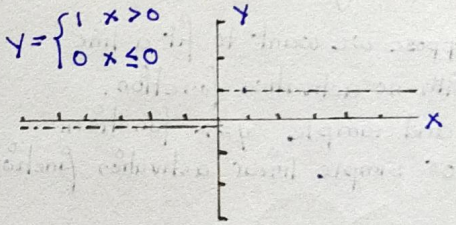
# CHAPTER 4 ACTIVATION FUNCTIONS

- The activation function is applied to the output of a neuron (or layers of neuron), which modifies the output.
- We use activation functions because if the activation function itself is non linear it allows for neural network with usually two or more hidden layers to map non linear functions.
- In general, neural network will have two types of activation function. The first will be activation function used in hidden layers and second will be used in the output layer. Usually, the activation function used for hidden neurons will be the same for all of them, but it doesn't have to.

## Step Activation function

In a single neuron, if weight x inputs + bias result in a value greater than 0, output is 1 otherwise, it will give output 0

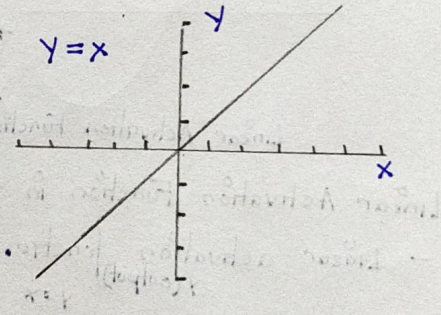
$$Y = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$




## Linear Activation Function

A linear function is simply the equation of line. It will appear as a straight line when graphed, where $y = x$ and the output is equal to the input.

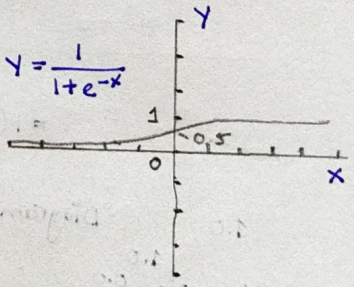
This activation function is usually applied to last layer's output in the case of a regression model.




## Sigmoid Activation Function

In step function, it is either (1) or (0).
It is hard to tell how close this function was to 1 or 0.
Maybe it is too close or too far.
Thus when it come to optimize weight and bias, its easier if activation function have more granular & informative.
And that function is Sigmoid Function.

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



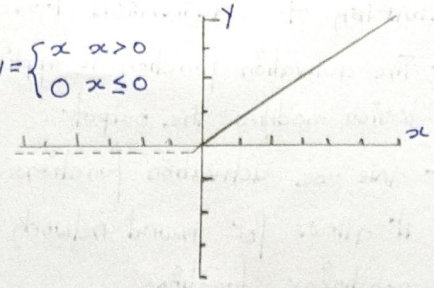

This function returns a value in the range of 0 for negative infinity, through 0.5 for input of 0. and to 1 for positive infinity

The output from sigmoid function, being in range from 0 to 1, also works better compared to the range of negative to positive infinity and adds non linearity.

## Rectified Linear Activation Function (ReLU)

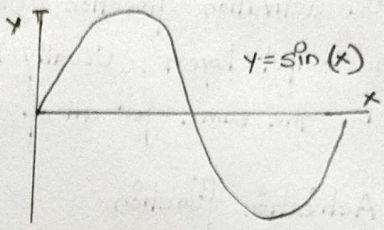
- ReLU is Rectifier Linear Units.
- It is $y=x$, clipped at 0 from negative side.
- If $x$ is less than or equal to 0, then $y$ is 0 otherwise $y$ is equal to $x$
- It also execute fast and efficient.
- Extremly close to being a linear activation function while remaining nonlinear, due to bend after 0.

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

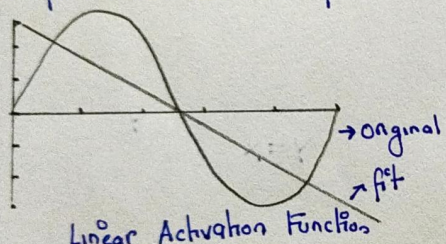



## Why Use Activation Function?

Non linear function –
A function cannot be represented well by straight line, such as sine function.



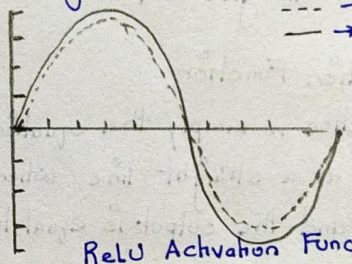

Suppose we want to fit a line with no activation function, and simple $y=x$ function. or simple linear activation function




Linear Activation Function

When using 2 hidden layers of 8 neuron each with ReLU activation function Following result is observed.

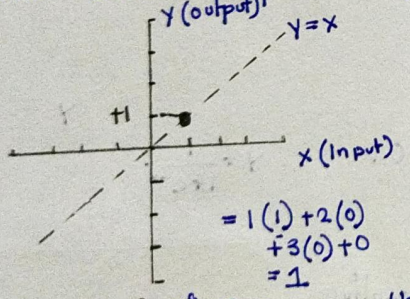
--- → fit.
— → original



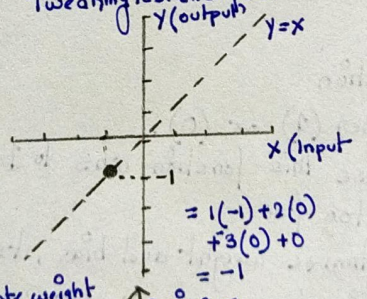
ReLU Activation Function

## Linear Activation Function in the Hidden Layers

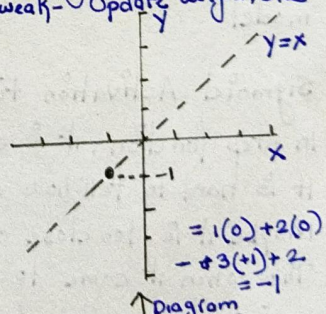
- Linear activation function $y=x$, and let's consider this single neuron level.



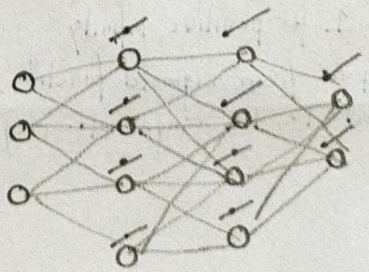

Tweaking last example




Tweak- Update weight, bias




No matter what this neuron weight and biases the output will be perfectly linear $y=x$ of the activation function.
In each layer in each neuron acts linearly, So entire network is a linear function as well.

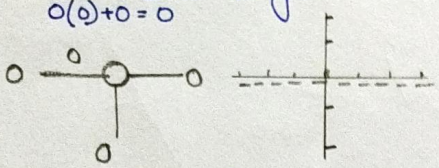


04.3 ReLU Activation in a Pair of Neurons $y = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$ [Refer to 04.5 first]

Lets begin again with a single neuron.

Zero weight

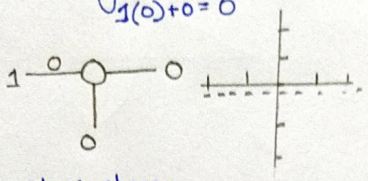
0(0)+0 = 0



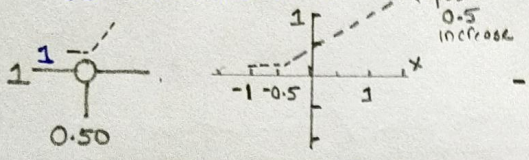
No matter what input we pass, output of this neuron will be 0 because weight is 0.

Now change input to 1

1(0)+0 = 0



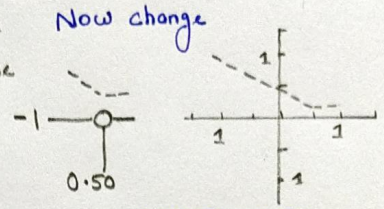
Set the bias to 0.50




1(1)+0.50 = 1.50

0.50 bias means in y axis line should pass through 0.5 and it should end at 1.

Now change

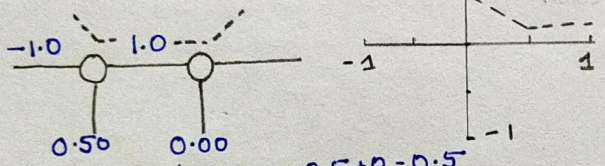


-1+0.50 = 0.5

Here line should pass through y = 0.5 and it should end at -1 (x axis)

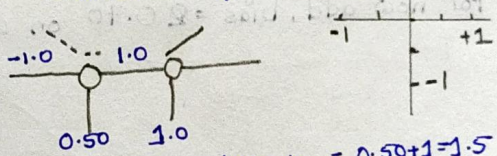
Here we can observe bias and weight influence slope of activation.

Let do with other example,



Line should pass through = 0.5+0 = 0.5 on y axis.
And should end at -1. Diagram is same because of first one because second neuron is 0.

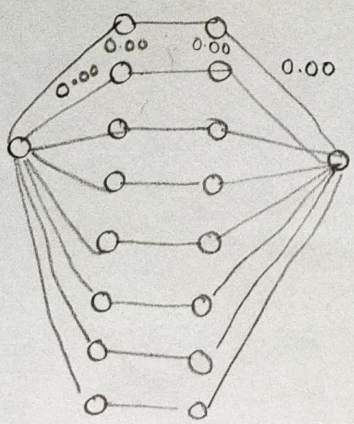
Another example,



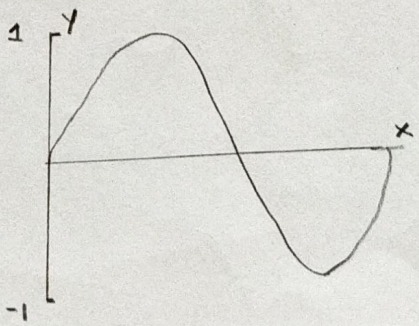
Line should pass through = 0.50+1 = 1.5 on y axis.
And should end at -1.

- Lets fit to the sine wave function using 2 hidden layers of 8 neuron each and we can fine tune the value to fit the curve.
- For simplicity we will assume layers are not densely connected, one neuron of layer connects to one layer of second. [That not the case of real models]
- Input is a single value and output a single value like the sine function. The output uses the linear activation function. & hidden layer will use ReLU.

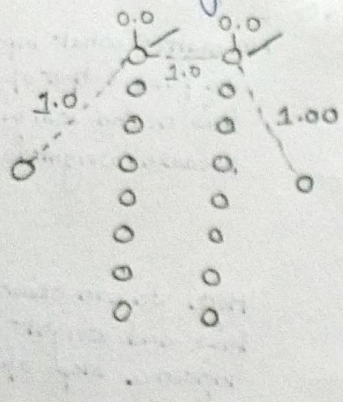
We will start with all weights = 0

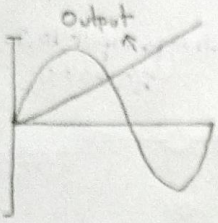



Expected Output

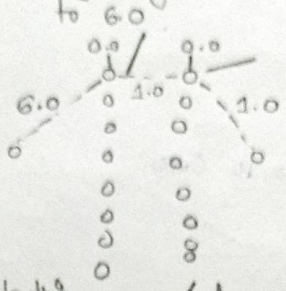


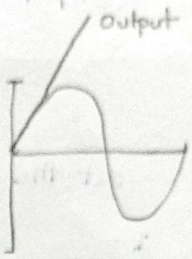
Now set the weight




In this case, we can see slop of overall function is impacted.
We can further increase slope by adjusting first neuron of first layer to 6.0



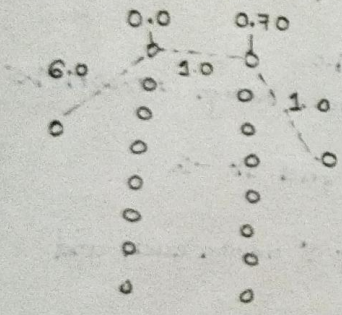



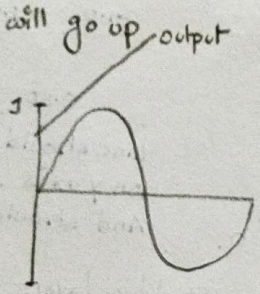

In this case / from — because as we increase the value it tilt straight

We can now see, inital slope of this function is what we like but we have a problem. Currently function never ends because this neuron pair never deactivates. So we need a deactivation point.

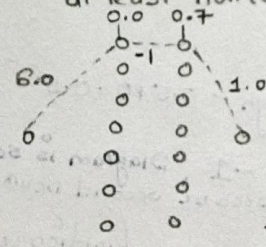
For now add, bias = 0.70 on 2nd neuron

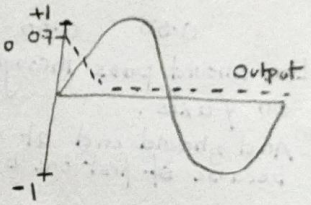
If bias we add +, it will go up




Now set weight for 2nd neuron to -1 causing a deactivation function at least horizontally



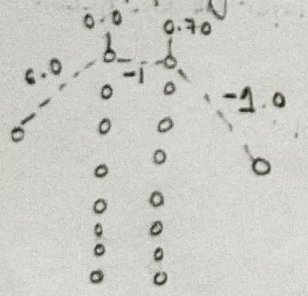



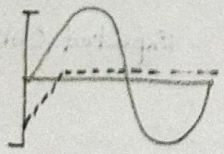

Here weight changes from 1 to -1 and that is why it flipped like this. bias = 0.7 so from y = 0.7 it will go down and become straight

Now we want to flip the slope back as we loose it will flipping.

So again flip output neuron from +1 to -1

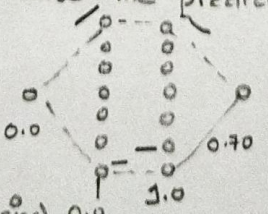
It will flip again

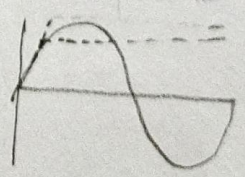


This is hand ophmization, so use

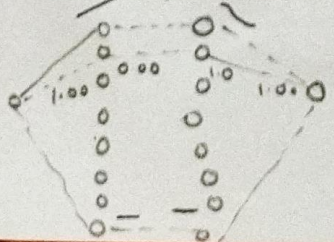
- Now we will take last neuron and raise the prediction

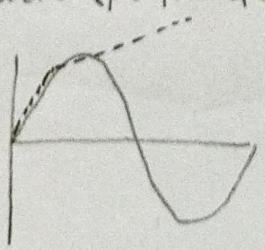


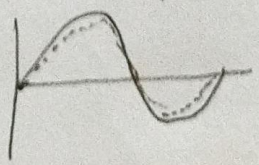
Now, completed first area of section (first phase of sine)

Set second pair of neuron



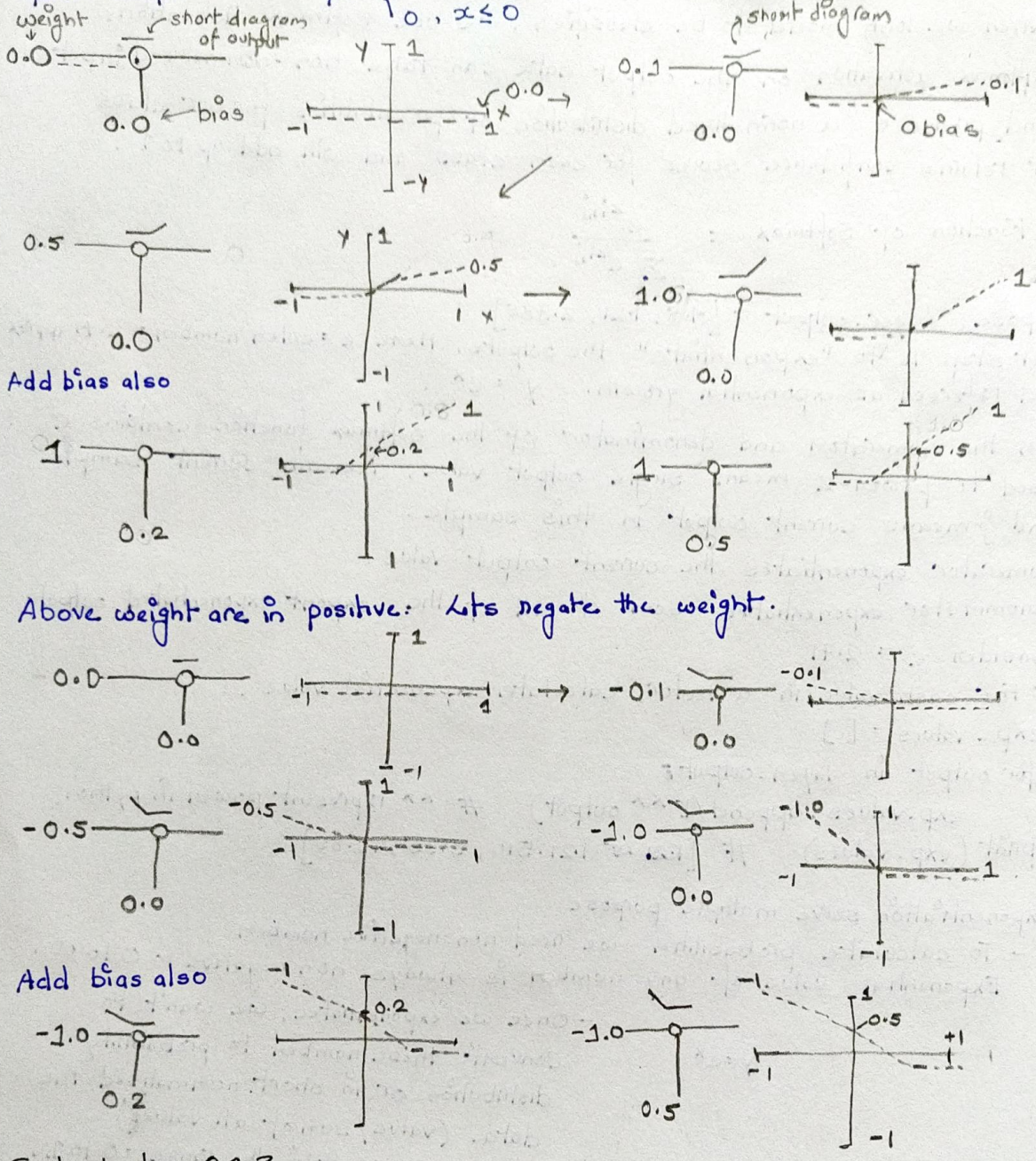



And slowly slowly we will tune all neuron and find the sine.



04-5

ReLU output. $y = \begin{cases} x, & x > 0 \\ 0, & x \le 0 \end{cases}$




Add bias also

Above weight are in positive: Lets negate the weight.




Add bias also




Go back to 04-3

# Softmax Activation Function

- When we look model to be classifier, we use Softmax Activation.
- Softmax activation on the output data can take non-normalized inputs and produces a normalized distribution of probabilities for our class It returns confidence scores for each class and will add up to 1.

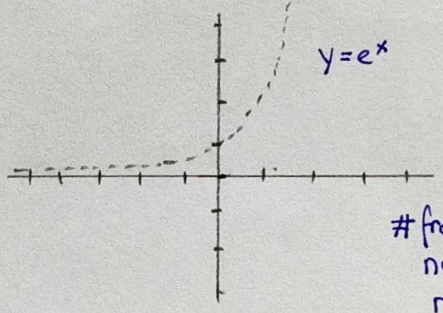
- Function of Softmax = $\frac{e^{z_{i,j}}}{\sum e^{z_{i,j}}}$

- Suppose, layer_output = [4.8, 1.21, 2.385]
  First step is to "exponentiate" the outputs. Here e = euler number = 2.71 approx
  And refereed as exponential growth. $y = e^x$.
- Both the numerator and denominator of the Softmax function contains e raised to power z means single output value, i means current sample and j means current output in this sample.
- Numerator exponentiates the current output value.
- Denominator exponentiates sum of all of the ~~expont~~ exponentiated output

consider e = 2.71

```
# For each value in a vector, calculate exponential value.
exp_values = []
for output in layer_output:
    exp_values.append(E ** output)   # ** represent power in python.
print(exp_values)  # [121.510, 3.35, 10.85]
```

- Exponentiation serve multiple purpose
  - To calculate probabilities we need non-negative number. Exponential value of any number is always non negative - 0 to ∞.




  - Once we exponentiated, we want to convert these number to probability distribution or in short normalised the data. (value / sum of all values) So this will result in the range - 0 to 1.

```
#from above exp_values.
norm_base = sum(exp_values)
norm_value = [].
for value in exp_values:
    norm_value.append(value / norm_base)
print(norm_values)  # [0.89, 0.024, 0.080]
print(sum(norm_values)) # 1.
```

This is the probability of output