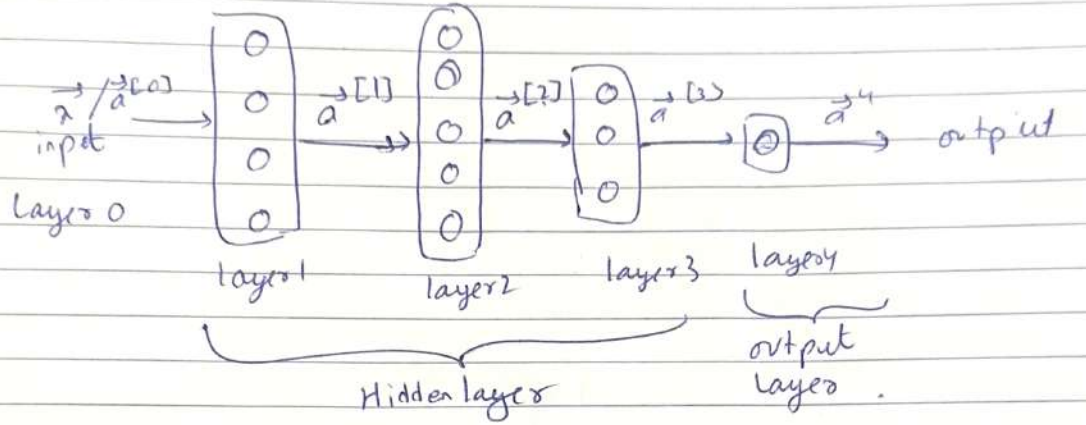
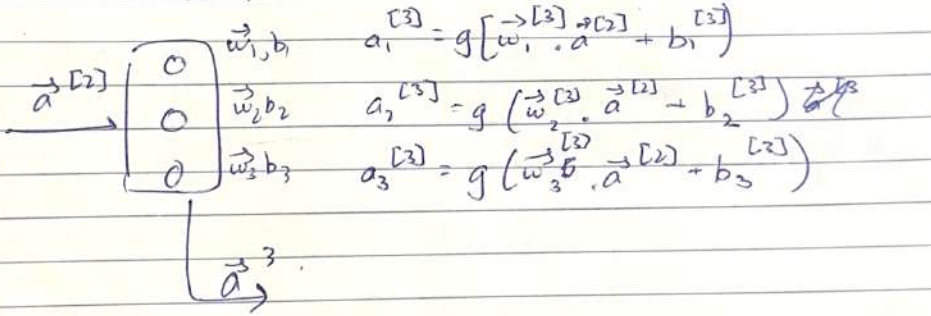
More complex neural network



Layer3 in detail.



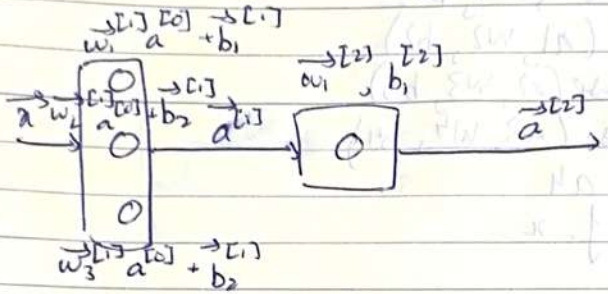
General eq.:- $a_j^{[l]} = g\left(\vec{w}_j^{[l]} \cdot \vec{a}^{[l-1]} + b_j^{[l]}\right)$

activation function results in activation values.

$l \rightarrow$ layer no

$j \Rightarrow$ neuron no.

Forward Prop – Theory



$x = np.array([200, 17])$

$a_1^{[1]} = g(\vec{w}_1^{[1]} \cdot \vec{x} + b_1^{[1]})$ $\quad$ $a_2^{[1]} = g(\vec{w}_2^{[1]} \cdot \vec{x} + b_2^{[1]})$

```
w1_1 = np.array([1, 2])
b1_1 = np.array([-1])
z1_1 = np.dot(w1_1, x) + b1_1
a1_1 = sigmoid(z1_1)
```

```
w1_2 = np.array([-3, 4])
b1_2 = np.array([1])
z1_2 = np.dot(w1_2, x) + b1_2
a1_2 = sigmoid(z1_2)
```

$a_3^{[1]} = g(\vec{w}_3^{[1]} \cdot \vec{x} + b_3^{[1]})$

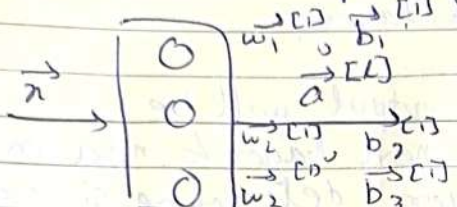
```
w1_3 = np.array([5, -6])
b1_3 = np.array([2])
z1_3 = np.dot(w1_3, x) + b1_3
a1_3 = sigmoid(z1_3)
```

```
a1 = np.array([a1_1, a2_1, a3_1])
```

$$a_1^{[2]} = g(\vec{w}_1^{[2]} \cdot \vec{a}^{[1]} + b_1^{[2]})$$

```
w2_1 = np.array([-7, 8, 9])
b2_1 = np.array([3])
z2_1 = np.dot(w2_1, a1) + b2_1
a2_1 = sigmoid(z2_1)
```

Forward Prop def in py



$\vec{w}_1^{[1]} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ $b_1^{[1]} = -1$

$\vec{w}_2^{[1]} = \begin{bmatrix} -3 \\ 4 \end{bmatrix}$ $b_2^{[1]} = 1$

$\vec{w}_3^{[1]} = \begin{bmatrix} 5 \\ -6 \end{bmatrix}$ $b_3^{[1]} = 2$

```
w = np.array([[1, -3, 5],
              [2, 4, -6]])

b = np.array([-1, 1, 2])

a_in = np.array([-2, 4])        a^[0] = x
```

```
def dense(a_in, W, b):
    units = W.shape[1]          ## no. of columns
                                no. of neurons
    a_out = np.zeros(units)     #array of 0s
    for j in range(units):      #goes through each column
        w = W[:, j]             #gets value of column
        z = np.dot(w, a_in) + b[j]
        a_out[j] = g(z)
    return a_out
```

g(z) ← defined outside

```
def sequential(x):
    a1 = dense(x, W1, b1)
    a2 = dense(a1, W2, b2)
    a3 = dense(a2, W3, b3)
    a4 = dense(a3, W4, b4)
    y_pred = a4
    return y_pred
```

Model Training Steps

① Specify how to compute output given input x and parameters w, b
(Defining model)

$f_{\vec{w},b}(\vec{x}) = ?$

model = Sequential([
Dense(units = 25, activation = 'sigmoid')

② Specify loss and cost

$L(f_{\vec{w},b}(\vec{x}), y)$ ← 1 example

$J(\vec{w}, b) = \frac{1}{m} \sum_{i=1}^{m} L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)})$ ← entire model

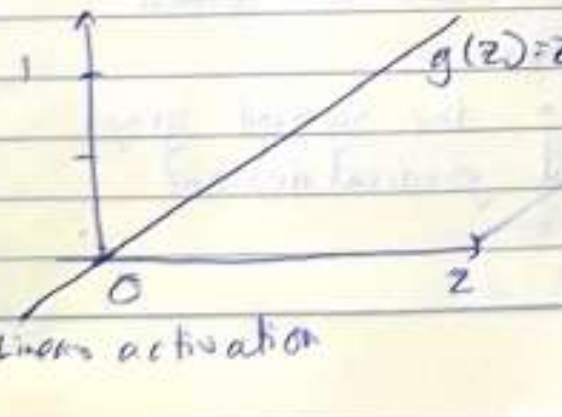
- model.compile(loss = BinaryCrossentropy())
  (loss = MeanSquaredError())

③ Train on data to minimize $J(\vec{w}, b)$

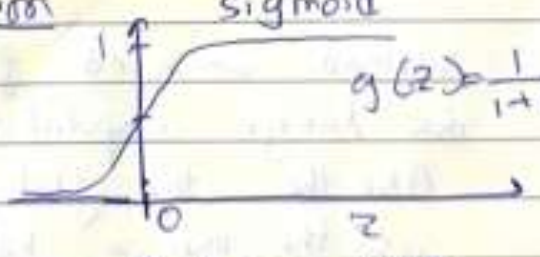
- model.fit(x, y, epochs = 100)

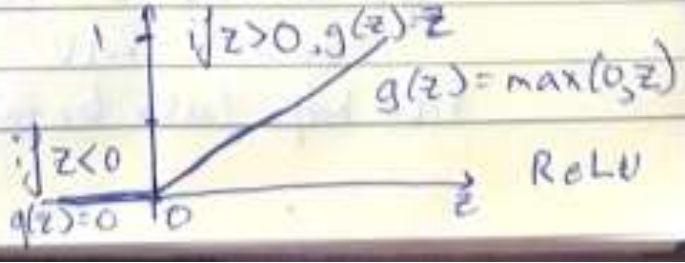
→ no of times to repeat/loop gradient descent

Different Activation function



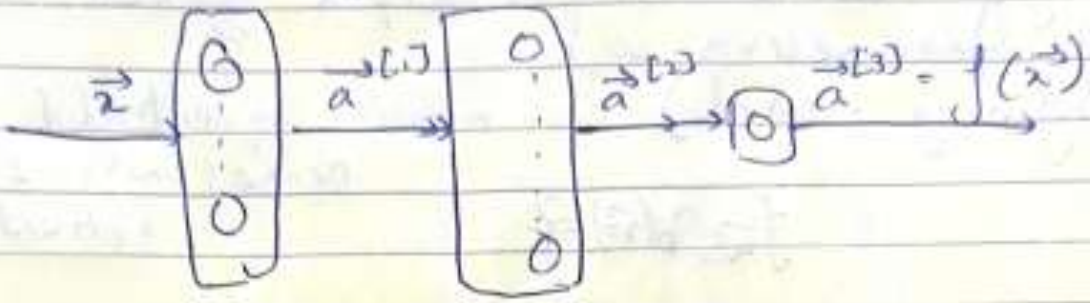
Linear activation





ReLU

Choosing Activation Function



Choosing for output layer?
- Depends on what type of values you want for 'y'
- For classification of 0/1 - Binary classification, we use sigmoid activation
- For some value prediction (+ve & -ve) we use 'linear' activation
- For some value prediction (+ve only), we use 'ReLU' activation
  ex. House Price.

Choosing for hidden layers?
- Most common choice is ReLU
  - Rectified Linear Unit.

Note: Sigmoid was used first but later not taken due large computational times.
Also the two flat ends in the sigmoid graph vs the one in Relu meant gradient descent is easier in ReLU
ReLU helps faster learning

Multi Class Classification
SOFTMAX

* 4 Possible outputs ($y = 1, 2, 3, 4$) :

X $z_1 = \vec{w}_1 \cdot \vec{x} + b_1$

$$a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y=1|\vec{x})$$

O $z_2 = \vec{w}_2 \cdot \vec{x} + b_2$

$$a_2 = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y=2|\vec{x})$$

□ $z_3 = \vec{w}_3 \cdot \vec{x} + b_3$

$$a_3 = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y=3|\vec{x})$$

Δ $z_4 = \vec{w}_4 \cdot \vec{x} + b_3$

$$a_4 = \frac{e^{z_4}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y=4|\vec{x})$$

$$a_1 + a_2 + a_3 + a_4 = 1$$

* Softmax Regression (N-possible outputs)

$y = 1, 2, 3, \ldots N$

$$z_j = \vec{w_j} \cdot \vec{x} + b_j \quad , \quad j = 1, 2, \ldots N$$

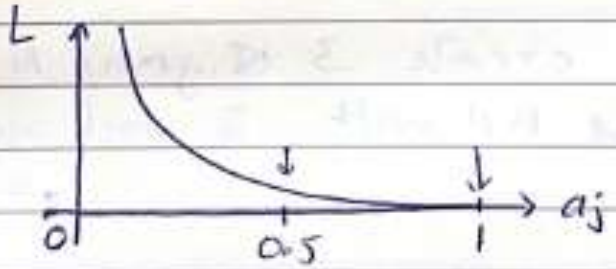
$$a_j = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}} = P(y = j \mid \vec{x})$$

$$a_1 + a_2 + \ldots + a_N = 1$$

*Cost

$$\text{loss}(a_1, \ldots, a_N, y) = \begin{cases} -\log a_1 & \text{if } y = 1 \\ -\log a_2 & \text{if } y = 2 \\ \vdots & \\ -\log a_N & \text{if } y = N \end{cases}$$

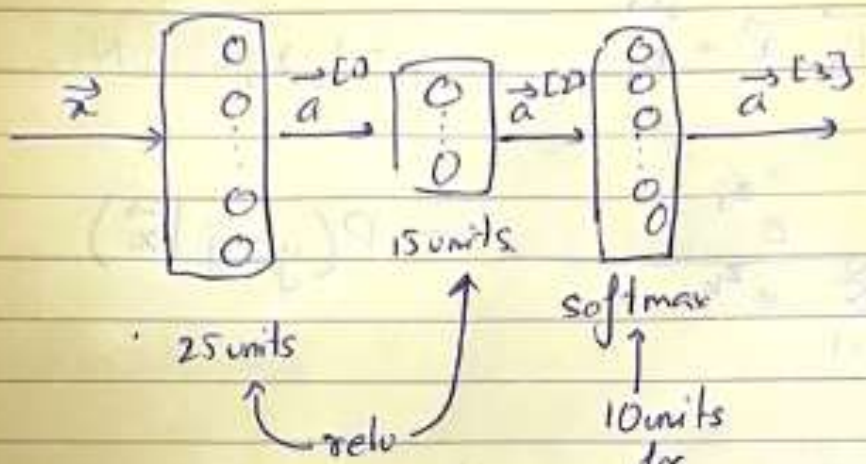
Ex. $\text{loss} = -\log(a_j)$ if $y = j$



Note: The value with the most chance is taken as output if the value of the output is small, then the loss function ~~takes~~ promotes the output to be as close to 1

Thus loss function tries to give more accurate results

Neural Network with Softmax output



← each node is one class

$$z_1^{[3]} = \vec{w}_1^{[3]} \cdot \vec{a}^{[2]} + b_1^{[3]} \qquad a_1^{[3]} = \frac{e^{z_1^{[3]}}}{e^{z_1^{[3]}} + \cdots e^{z_{10}^{[3]}}} = P(y=1 \mid \vec{x})$$

$$z_{10}^{[3]} = \vec{w}_{10}^{[3]} \cdot \vec{a}^{[2]} + b_{10}^{[3]} \qquad a_{10}^{[3]} = \frac{e^{z_{10}^{[3]}}}{e^{z_1^{[3]}} + \cdots + e^{z_{10}^{[3]}}} = P(y=10 \mid \vec{x})$$

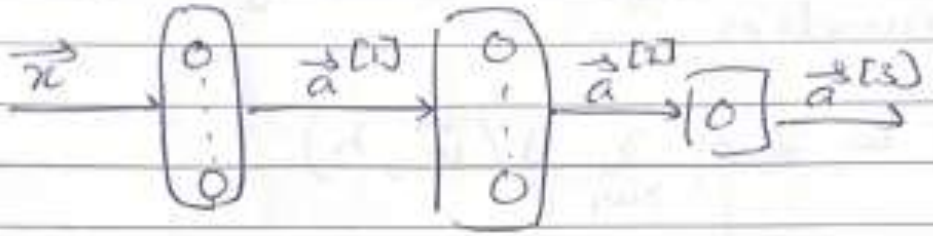
Multilabel Classification

either create 3 Sigmoid NN

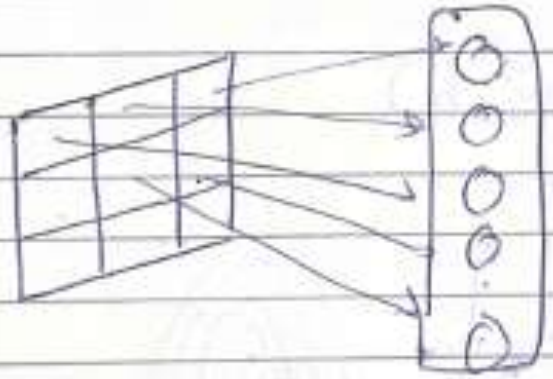
or one NN with 3 unit sigmoid output

Additional Layer Types

Dense layer (Uphi Now) → Each neuron is a function of all activation outputs of the previous layer.



Convolutional Layer → Each ~~neural~~ neuron only looks at part of the previous layer's output



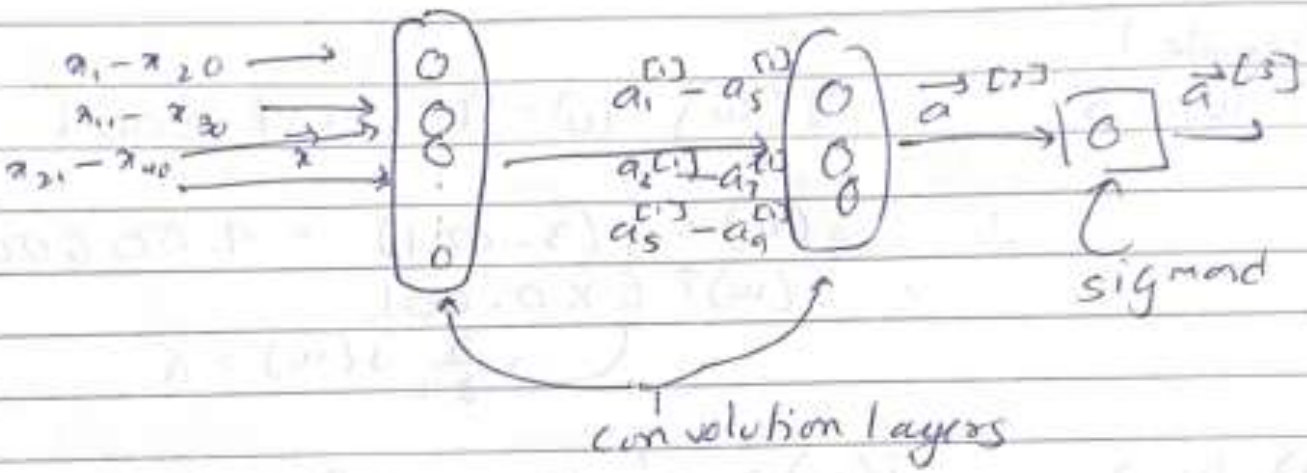
↳ This makes computation faster

↳ less training data needed

↳ less prone to overfitting

Ex:

x1 x2 x3 . . . . . .



The parameters we need to keep in mind in input window for 1 neuron & how many neurons should a layer have