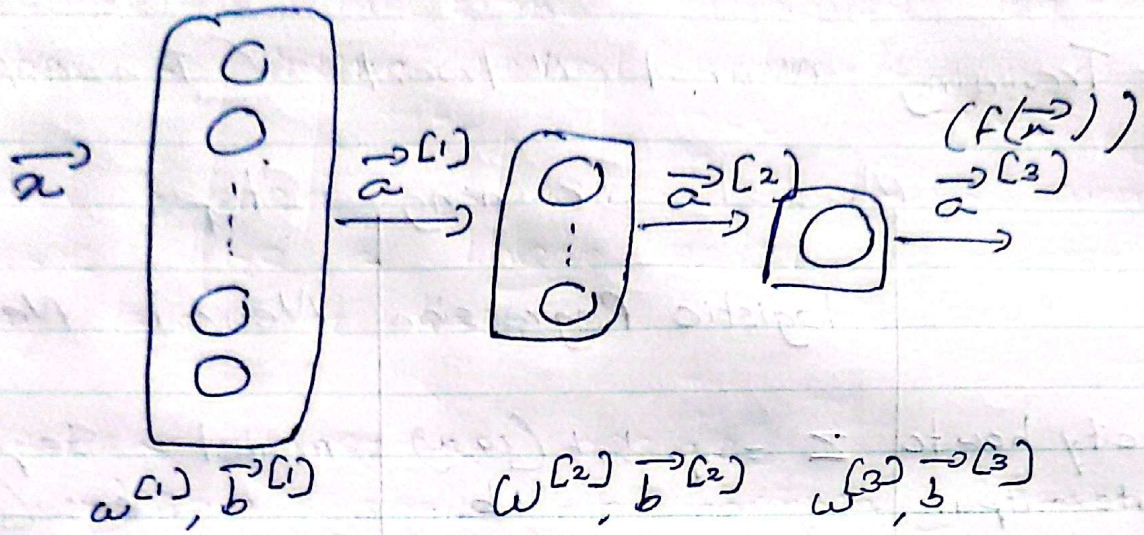
Week 02

01] Training Neural Network (Tenserflow)

Model Training Steps

	logistic Regression	Neural Network
① Specify how to Compute output given input x and parameteres w, b (define model) $f_{\vec{w},b}(\vec{x}) = ?$	$z = np.dot(w,x) + b$ $f\_x = 1/(1+np.exp(-z))$	model = Sequential([Dense(....) Dense(...) Dense(....)])
② Specify loss and cost $L(f_{\vec{w},b}(\vec{x}), y)$ $J(\vec{w}, b) = \frac{1}{m}\sum_{i=1}^{m} L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)})$	logistic loss loss = -y*np.log(f_x) - (1-y)*np.log(1-f_x)	binary cross entropy model.compile(loss = BinaryCrossentropy)
③ Train on data to minimize $J(\vec{w}, b)$	$w = w - alpha * dj\_dw$ $b = b - alpha * dj\_dw$	model.fit(X, y, epochs=100)

1] Create the model




```
import tenserflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Segential ([
    Dense(units = 25, activation = 'sigmaid'),
    Dense(units = 15, activation = 'sigmaid'),
    Dense(units = 1, activation = 'sigmaid'), ])
```

2] Loss and Cost functions

i) binary Classification

$$L(f(\vec{x}), y) = -y \log(f(\vec{x})) - (1-y) \log(1 - f(\vec{x}))$$

*) Compare prediction vs target

*) Logistic loss also known as binary cross entropy

```
from tensorflow.keras.losses import
BinaryCrossentropy, MeanSquaredError

model.compile(loss = BinaryCrossentropy())
```

ii) Regression (predicting numbers and not categories)

```
model.complie(loss = MeanSquaredError())
```

3) Gradient descent

repeat {

$$w_j^{[L]} = w_j^{[L]} - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$$

$$b_j^{[L]} = b_j^{[L]} - \alpha \frac{\partial}{\partial b_j} J(\vec{w}, b)$$

}

4) Compute derivatives for gradient descent using "back propagation"

```
model.fit(X, y, epochs = 100)
```

02] Chosing activation functions

*) Activation functions introduce non-linearity in to a neural network.

*) Without them, a neural network would behave like a linear model, no matter how many layers it has.

Activation functions decide

*) how signals flow forward (forward propergation)

*) How gradients flow backward (back propergation /during learning)

Common Activation Functions

1] ReLU (Rectified Linear Unit)

$$f(z) = \max(0, z)$$

Derivative :

$$f'(z) \begin{cases} 1 & z > 0 \ (0, \infty) \\ 0 & z \leq 0 \end{cases}$$

*) Very fast to compute
*) Avoids vanishing gradients (for positive inputs)
*) Outputs only positive values
*) Neurons can die if outputs become 0 for all inputs (then gradient = 0)

2] Sigmoid

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

Derivative:

$$f'(z) = f(z) \times (1 - f(z))$$

*) Output Range (0, 1)
*) Smooth, S shaped curve
*) Output can be interpreted as probability
*) Causes vanishing gradient when used in hidden layers
*) Gradients get tiny when z is far from 0

3] Linear

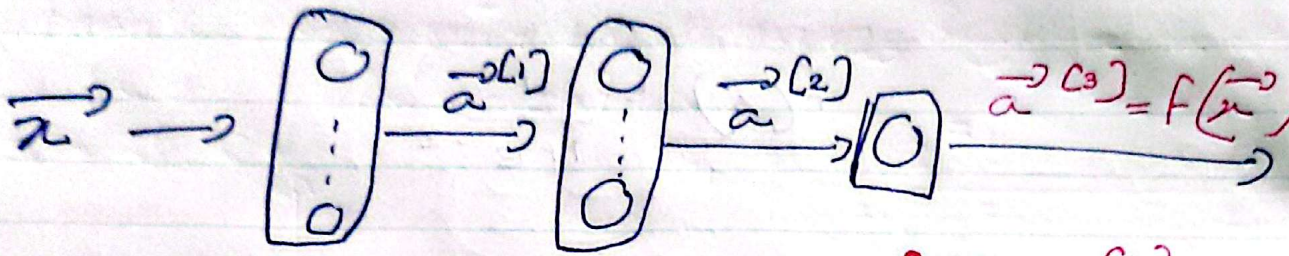
$$f(z) = z$$

Derivative:

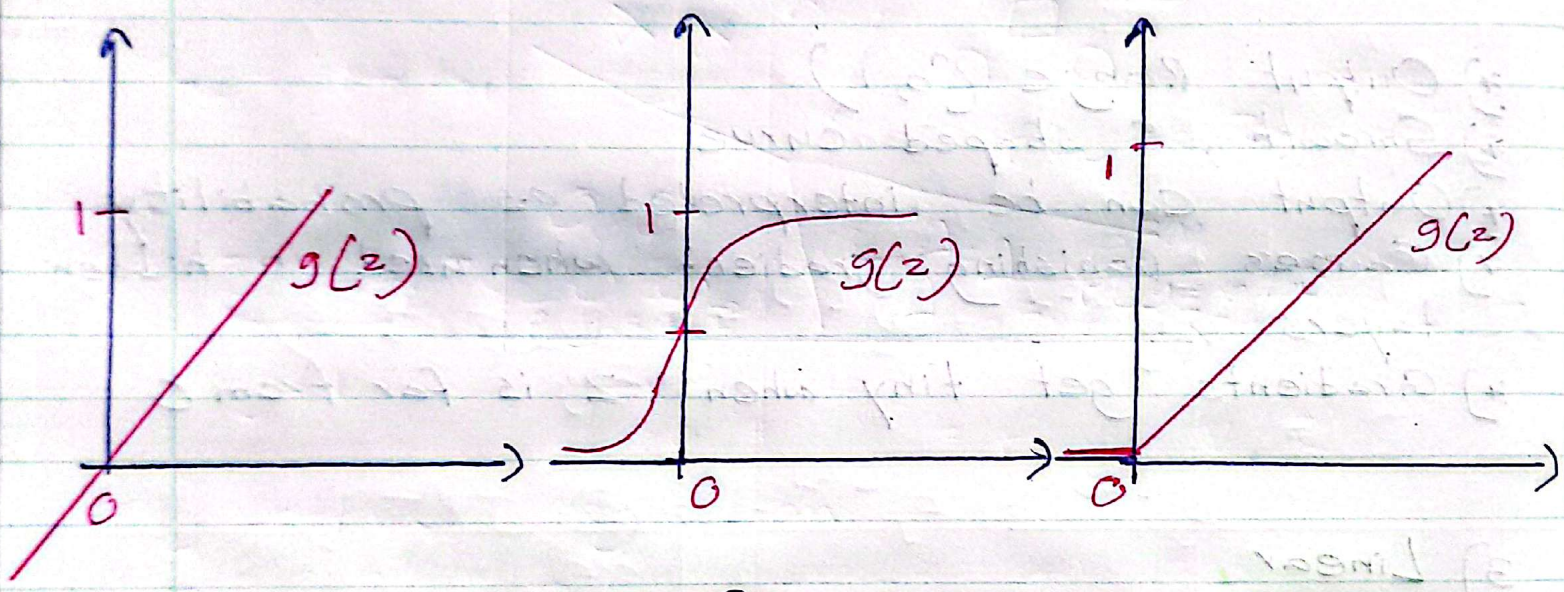
$$f'(z) = 1$$

*) No non-linearity (No activation function)
*) Output range $(-\infty, \infty)$

3) Output layer



$f(\vec{x}) = a_1^{[3]} = g(z_1^{[3]})$



Regression
Linear activation
$y = +/-$

Binary Classification
Sigmoid activation
$y = 0/1$

Regression
RELU
$y = 0$ or $+$

4) Hidden layer

*) Most Common choice is ReLU because it avoids vanishing gradient, fast and effective

Vanishing gradient

*) As gradients are propagated backward through many layers, they can become smaller and smaller (approaching zero)

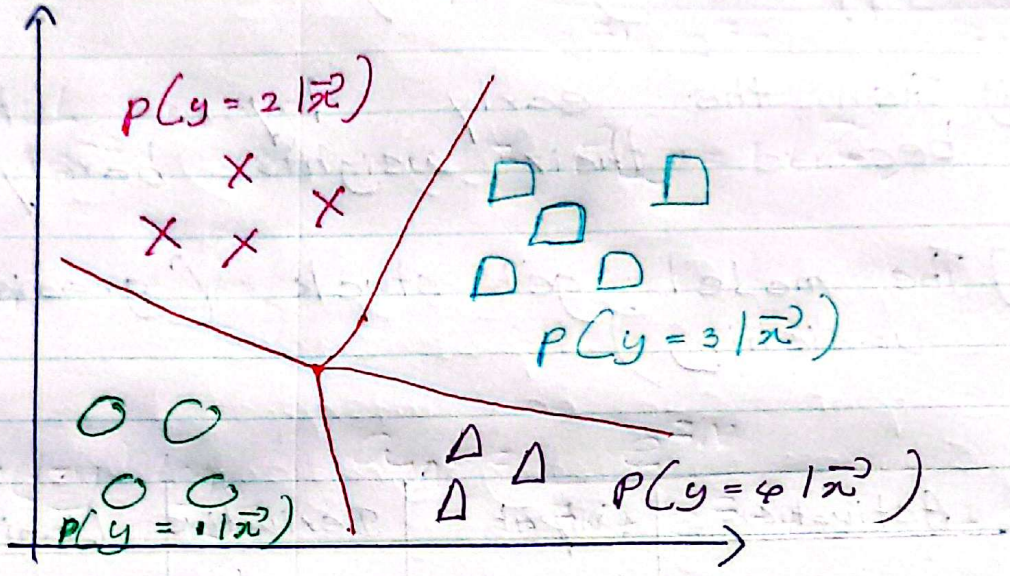
*) Then the early layers stop learning because their weights barely change

*) The model get stuck (gradient descent vanishing)

Activation	Output	Derivative	Vanishing Gradient descent
Sigmoid	(0, 1)	(0, 0.25)	High
ReLU	(0, ∞)	(0, 1)	Low
Tanh	(-1, 1)	(0, 1)	medium

03] Multi Class

*) If target y can take on more than two possible values it a multi class classification



04] Softmax regression

② Logistic regression (2 possible output values)

○ $a_1 = g(z) = \frac{1}{1+e^{-z}} = P(y=1|\vec{x})$

× $a_2 = 1 - a_1 = P(y=0|\vec{x})$

Softmax regression (4 possible outputs)

$y = 1, 2, 3, 4$

○ $z_1 = \vec{w_1} \cdot \vec{x} + b_1$ $\quad a_1 = \dfrac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$

○ × □ △

$= P(y=1 \mid \vec{x})$ 0.30

× $z_2 = \vec{w_2} \cdot \vec{x} + b_2$ $\quad a_2 = \dfrac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$

$= P(y=2 \mid \vec{x})$ 0.20

□ $z_3 = \vec{w_3} \cdot \vec{x} + b_3$ $\quad a_3 = \dfrac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$

$= P(y=3 \mid \vec{x})$ 0.15

△ $z_4 = \vec{w_4} \cdot \vec{x} + b_4$ $\quad a_4 = \dfrac{e^{z_4}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$

$= P(y=4 \mid \vec{x})$ 0.35

Softmax regression (N possible outputs)

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

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

$(a_1 + a_2 + \cdots + a_N = 1)$

Cost

Logistic Regression

$$z = \vec{w} \cdot \vec{x} + b$$

$$a_1 = g(z) = \frac{1}{1+e^{-z}} = P(y=1 | \vec{x})$$

$$a_2 = 1 - a_1 = P(y=0 | \vec{x})$$

$$\text{loss} = \underbrace{-y \log a_1}_{\text{if } y=1} \underbrace{-(1-y) \log (1-a_1)}_{\text{if } y=0}$$

$$J(\vec{w}, b) = \text{average loss}$$

Softmax Regression

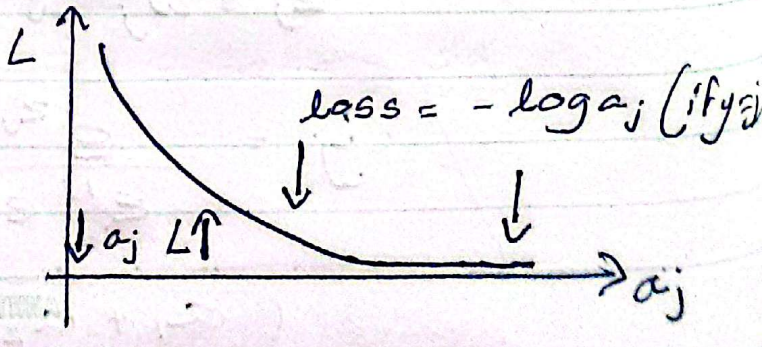
$$a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + \dots e^{z_N}} = P(y=1 | \vec{x})$$

$$\vdots$$

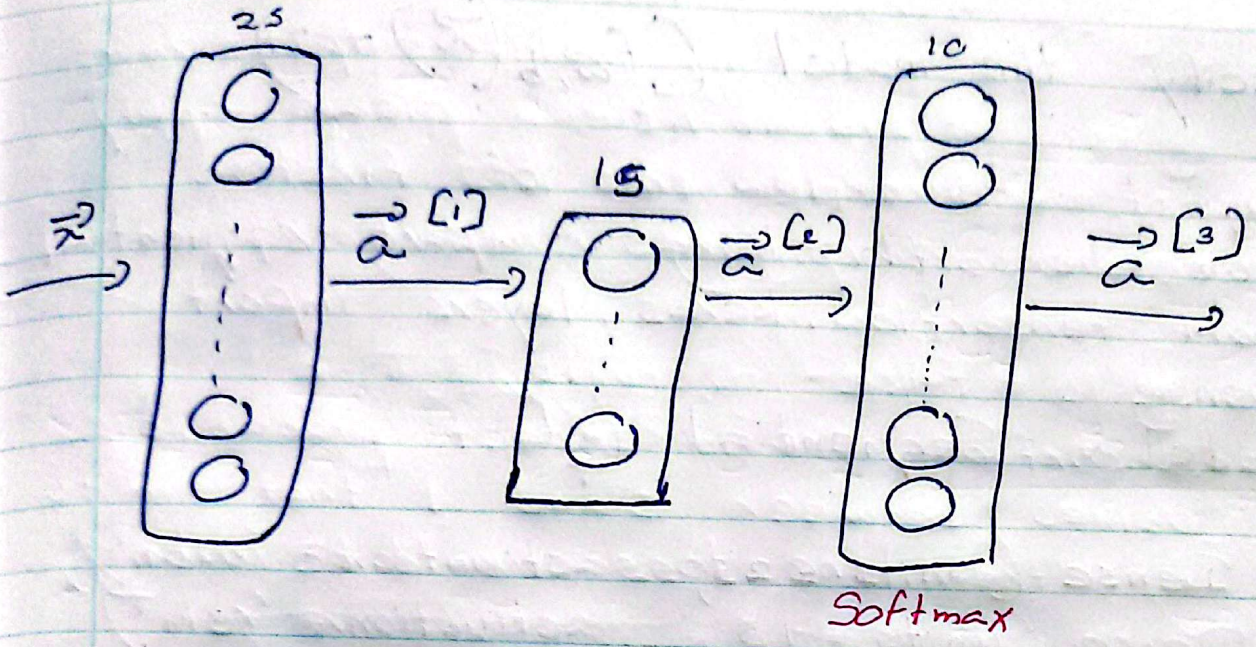
$$a_N = \frac{e^{z_N}}{e^{z_1} + e^{z_2} + \dots + e^{z_N}} = P(y=N | \vec{x})$$

Crossentropy loss

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



05] Neural Network with Softmax output



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

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

logistic regression :

$$a_1^{[3]} = g(z_1^{[3]}) \qquad a_2^{[3]} = g(z_2^{[3]})$$

Soft max regression

$$\vec{a}^{[3]} = (a_1^{[3]} \dots a_{10}^{[3]}) = g(z_1^{[3]}, \dots z_{10}^{[3]})$$

Default Version (Not recommended)

1) Specify the model ($f_{\vec{w},b}(\vec{x}) = ?$)

```
import tensorflow as tf
from tensorflow.keras import sequential
from tensorflow.keras.layers import
Dense
model = Sequential([
    Dense(units = 25, activation = 'relu'),
    Dense(units = 15, activation = 'relu'),
    Dense(units = 10, activation = 'softmax')
])
```

2) Specify loss and Cost ($L(f_{\vec{w},b}(\vec{x}), y)$)

```
from tensorflow.keras.losses import
SparseCategoricalCrossentropy

model.compile(loss = SparseCategorical
                     Crossentropy())
```

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

```
model.fit(X, Y, epochs = 100)
```

06] Improved Implementation of Softmax

```
model = Sequential([
    Dense(units = 25, activation = 'relu'),
    Dense(units = 15, activation = 'relu'),
    Dense(units = 10, activation = 'linear')])

from tenserflow.keras.losses import
    SparseCategoricalCrossEntarpy

model.compile(...., loss = SparseCategorical
                           Cross entropy
                           (from_logits = True))

model.fit(X, Y, epochs = 100)

logits = model(X)
f_X = tf.nn.softmax(logits)
```

*) In neural networks, before applying an activation function the output of the last linear layer is called the logit

*) If you apply an activation you will get probabilities (a).

*) If you don't apply you just have the raw scores (z)

z = logits

a = activated outputs (probabilities)

*) from_logits = True :
This tells us my model output raw logits, not probabilities, please apply Softmax inside the loss function before computing cross entropy

*) from_logits = False (Default):
Tensorflow would expect probabilities (values between 0 and 1 that sum to 1), then the loss would be wrong

Numerical round error

*) You set the output layer 'activation = linear' because you already applying softmax hadled by loss function, internally, if you set activation = softmax it will double softmax and gives probabilities not raw outputs

*) After training we want to see actual probabilities or class predictions, since we use linear as the output layer it gives logits not probabilities,

Sowe → Softmax(logits) to take the probabilities

07] Multi label Classification

*) In multi label classification, each input can belong to more than one class simultaneously

One input → multiple correct Classes

example:

*) movie genre → Action, comedy both

*) multi class classification means each input belongs to exactly one class out of many possible classes.

One input → One Correct class

*) In multi label classification, output layer consists of one neuron per label

*) Use sigmoid function for each neuron (independent binary output)

*) Use binary cross entropy as loss

*) Each neuron outputs probability between 0 and 1, independently.

08] Adam Algorithm for optimization

*) Adam : Adaptive Moment Estimation
(not just one α)

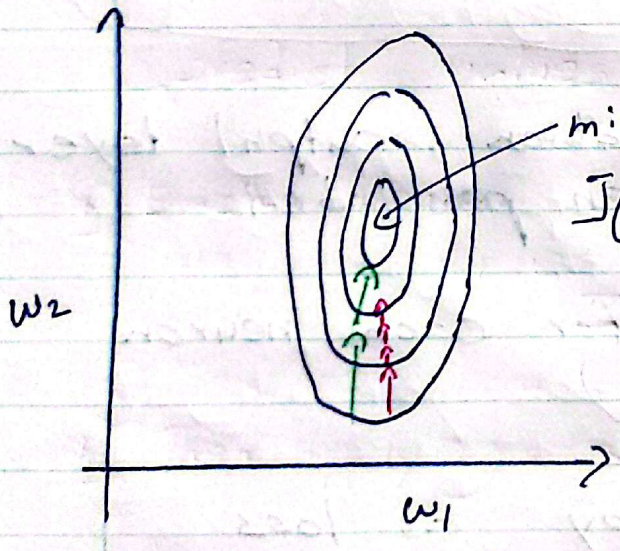
$$w_1 = w_1 - \alpha_1 \frac{\partial}{\partial w_1} J(\vec{w}, b)$$

$$\vdots$$

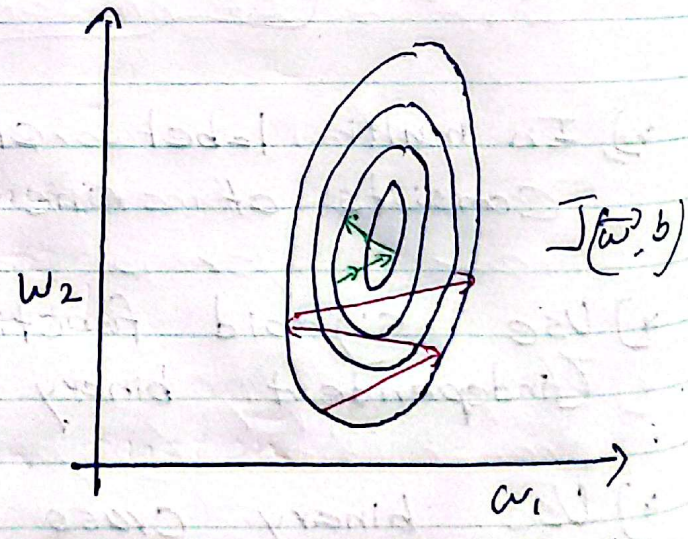
$$w_{10} = w_{10} - \alpha_{10} \frac{\partial}{\partial w_{10}} J(\vec{w}, b)$$

$$b = b - \alpha_{11} \frac{\partial}{\partial b} J(\vec{w}, b)$$

*) Faster and efficient than gradient descent



If w_j (or b) keeps moving in same direction, increase α_j



If w_j (or b) keeps oscillating reduce α_j

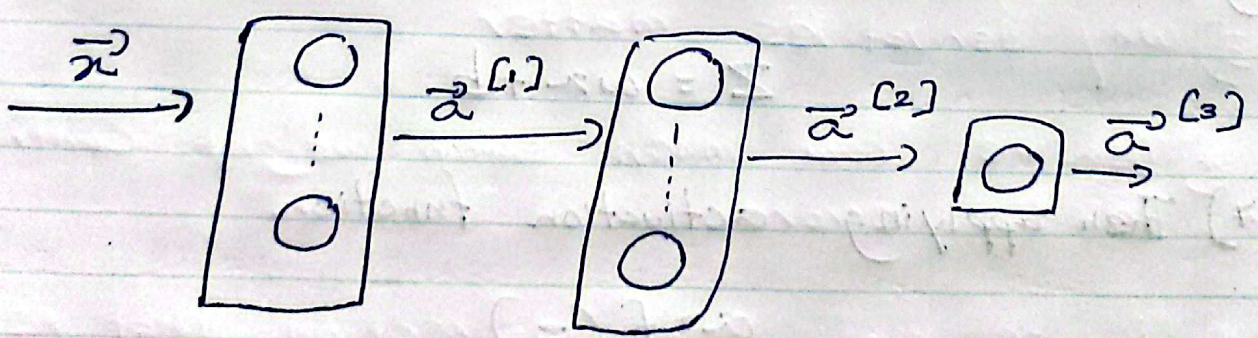
```
model = Sequential([
    tf.keras.layers.Dense(units=25, activation='sigmoid'),
    tf.keras.layers.Dense(units=15, activation='sigmoid'),
    tf.keras.layers.Dense(units=10, activation='linear')
])

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True))

model.fit(X, Y, epochs=100)
```

09] Layer Types

Dense layer



*) Each neuron output is a function of all the activation outputs of the previous layer

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

Convolutional Layer

*) Each neuron only looks at part of the previous layer's outputs.

Why ?

-) Faster Computation
-) Need less training data (less prone to overfitting)

10) How neural network works :

1) Forward propagation

*) This happens when data (inputs) flow forward through the network : from input → hidden layers → output

(predict values)

*) Each neuron does this :

$$Z = wx + b$$

*) Then applying activation function

$$a = f(z)$$

2] Backpropagation

*) This is how the network learns :- by calculating how wrong the prediction was (error) and update weights to reduce it.

Based on Calculs (Chain rule)

Step 1 : Compute loss

$$L = -(y \log(\hat{y}) + (1-y) \log(1-\hat{y}))$$

y => true label
$\hat{y}$ => predicted output (from forward propagation)

Step 2 : Compute Derivative of Loss (Error Signal)

$\frac{\partial L}{\partial w}$ (This tells us how much changing w changes the loss)

Step 3 : Apply Chain rule

*) we propagate the error backwards layer by layer using chain rule

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$$

- $\frac{\partial L}{\partial a}$: how loss changes with activation
- $\frac{\partial a}{\partial z}$: $f'(z)$ derivative of activation
- $\frac{\partial z}{\partial w}$: derivative of weighted sum

Step 4: Update weights

After Computing gradients, we update weights using optimization (Gradient descent, Adam) to reduce the loss.

$$w = w - \alpha \frac{\partial L}{\partial w}$$

ex :

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

$$L = -[y \log(\hat{y}) + (1-y) \log(1-\hat{y})]$$

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w}$$

$$\frac{\partial L}{\partial w} = (\hat{y} - y) \cdot x$$

3) Why derivatives matter

*) Measure how much each weight contributed to the error

*) Update only in the direction that reduces error

*) Learn patterns through repitition over many examples (epochs)