Semester	VII	L	T	P	C	COURSE CODE
Regulation	V20	3	0	0	3	V20CSTPE11
Name of the Course	Deep Learning (Professional Elective-III)					
Branch	Common to CSE & CST					

Syllabus Details

Course Outcomes: After Successful completion of the Course, the student will be able to:

CO1: Describe the fundamentals of deep learning. **(K2)**

CO2: Illustrate the working of deep feed forward neural networks. **(K2)**

CO3: Discuss regularization and optimization techniques used in deep neural networks. **(K2)**

CO4: Illustrate the working of convolution neural networks. **(K2)**

CO5: Explain about recurrent and recursive neural networks. **(K2)**

UNIT-I: Introduction: Historical Trends in Deep Learning, The Many Names and Changing Fortunes of Neural Networks, Increasing Dataset Sizes, Increasing Model Sizes, Increasing Accuracy, Complexity and Real-World Impact.

UNIT-II: Deep Feed forward Networks: Learning XOR, Gradient-Based Learning, Hidden Units, Architecture Design, Back Propagation and Other Differentiation Algorithms.

UNIT-III: Regularization for Deep Learning: Parameter Norm Penalties, Early Stopping, Dropout; **Optimization for Training Deep Models:** How Learning Differs from Pure Optimization, Challenges, Basic Algorithms, Parameter Initialization Strategies, Algorithms with Adaptive Learning Rates, Optimization Strategies and Meta-Algorithms.

Adam, RMSProp

UNIT-IV: Convolution Networks: The Convolution Operation, Motivation, Pooling, Convolution and Pooling as an Infinitely StrongPrior, Variants of Basic Convolution Functions, Structured Outputs, Data Types, Efficient Convolution Algorithms, Random or Unsupervised Features, The Neuroscientific Basis for Convolutional Networks, Convolutional Networks and the History of Deep Learning.

CNN
RNN

UNIT-V: Sequence Modeling- Recurrent and Recursive Nets: Unfolding Computational Graphs, Recurrent Neural Networks, Bidirectional RNNs, Encoder-Decoder Sequence-to-Sequence Architectures, Deep Recurrent Networks, Recursive Neural Networks, The Challenge of Long-Term Dependencies, Echo State Networks, Leaky Units and Other Strategies for Multiple Time Scales, LSTM and Other Gated RNNs, Explicit Memory.

Textbooks:

1. Deep Learning, Ian Goodfellow, YoshuaBengio, and Aaron Courville, MIT Press.

Reference Books:

1. Neural Networks and Deep Learning, Charu C. Aggarwal, Springer.
2. Fundamentals of Deep Learning, Nikhil Buduma, 1st Edition, O'Reilly

AI, ML, DL

⊛ **AI** :- It is the broad idea that machines can intelligently execute tasks by mimicking human behaviors;

— AI Recreates human intelligence & behavior using algorithms, data & models ✓

— Ex :- self-driving cars, security systems & medical imaging; Siri & Alexa ---

⊛ **ML** :- subset of AI, revolves around the idea that machines can learn & adapt through experiences & data to complete specific tasks; ✓

— ML is an AI algorithm which allow system to learn from data;

Ex — sentiment analysis, predictive maintenance, spam filters --

structured data :- data arranged in rows & columns

Ex :- Tables, excel sheets, databases

⊛ **DL** :- DL attempts to mimic the human brain through neural networks;

— DL is the evolution of machine learning & NN's which uses advanced computer programming & training to understand complex patterns hidden in large data sets; Ex :- image classification, NLP, generative models ✓

— DL is about understanding how the human brain works in different situations & then trying to recreate its behavior;

— used to complete complex tasks & unstructured data → email, social media posts, audio's, images, videos ---

Fun prompt :- Drawing a cat :-

AI — you tell the robot, "draw a cat", & it tries using rules you give it. It acts smart but doesn't really learn on its own

ML — you show the robot 100 pics of cats & it learns from them to draw its own cat ✓

DL — you show it 10,000 pics, & it uses a deep brain-like system to learn & very accurate ---

Just like a wrapper around Tensorflow ✓

- keras, Tensorflow, pytorch; (Libraries)
- mathematics + statistics; problem solving;
- GPU = for heavy DL projects; (NVIDIA) — Titan RTX (model)
- Why DL Popular :- ① Data growth (volume of data)
  - ② Hardware advancements — GPU, TPU (Tensorflow processing unit) (google)
  - ③ python & opensource ecosystems;
  - ④ frameworks :- pytorch (2016) (facebook), Tensorflow 2.0 (2015) ✓ (google), Dist Belief (2011) (google)
  - ⑤ Cloud & AI Boom;
  - ⑥ High accuracy with large data ⑦ automatic feature extraction
  - ⑧ versatility across domains — CNN, RNN, Transformers, audio, autonomous vehicles
  - ⑨ Breakthroughs in AI Capacities — ChatGPT, Deepfake, AlphaGo;
  - ⑩ state of art performance & scalable;

(keras + Tensorflow) ✓ industry ① (best)

pytorch → research ✓

Snippet :-
```
from tensorflow import keras
model = keras.Sequential([keras.layers.Dense(10, input_shape=(784,), activation='sigmoid')])
```

Application ↓
- Self-driving cars;
- Gameplaying agents;
- Image colorization;
- Pixel Restoration;

Important terms :-

① Neuron, Perceptron;

② Activation functions :-
- ReLu (Rectified Linear unit)
- (logit) sigmoid, Tanh, softmax;

Sigmoid $(x) = \frac{1}{1+e^{-x}}$ $e = 2.71828$

sigmoid func converts input into range 0 & 1 ✓

③ layers :- I/P, H/L, O/P;

I/p = Raw data, H/L = Perform transformations, O/p = final prediction ✓

④ Loss functions — measures error b/t P & A O/P
- MSE = Regression
- Cross-entropy = classification;

⑤ Backpropagation — Its a algorithm to minimize (loss) by updating weights using gradient;

⑥ Gradient Descent — used to update weights to reduce errors;

(Optimization) → stochastic gradient descent, Adam, RMS prop;

model architectures :-

① ANN, CNN, RNN ② Transformers

Features	ANN (Artificial NN)	CNN (Convolutional NN)	RNN (Recurrent NN) → LSTM, NLP
Purposes	General purpose NN	mainly for images & Spatial data	Sequence & time-series data
Input type	fixed size Vector	images	text, audio, time-
use cases	tabular data, basic classification;	Images, video, object detection;	Language modelling, speech, time prediction (LSTM/GRU)
Real world examples	creditcard fraud detection, medical diagnosis; stock market forecasting;	face Recognition, object detection; medical imaging, satellite images Analysis;	google Translate audio to text Analysis tweets, Review weather forecast
For	Regression/Classification	Computer vision/images	Time series & Analysis

BERT

time, effort, money → for selection of a particular architecture ✓

image classification → ResNET
Text classification → BERT
Image segmentation → UNet
Image translation → Pix 2 Pix
object detection → YOLO
speech generation → WaveNET

Readily available AI Architectures ✓

Training & performance :-

① Epoch ;

② Batch size ;

③ overfitting & underfitting ;

④ Regularization :- prevent overfitting ; L_1/L_2 Regularization, dropout, early stopping

⑤ performance metrics :-

Classification	Regression
Accuracy	MSE
precision	RMSE
Recall	MAE
F1-Score	R^2 - Score

Input layer = where data is fed into network ;

hidden layer = where all the mathematical calculations are done using weights, biases & data

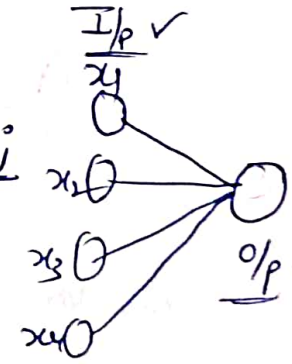
o/p layer = where meaningful results are obtained from

Activation func' — unit step func'

Neuron Vs perceptron :-

* perceptron :- Basic perceptron doesnot have a hidden layer ; (single layer)

→ only ① Input layer
② o/p layer ;



— solves only linearly seperable problems ; like AND, OR ;

(ii) multilayer perceptron :- (MLP) :- more complex patterns ;

i.e NN

— Includes one (or) more hidden layers

— Uses activation functions ;

Note :- MLP is a subset of NN's ; i.e ANN is the umbrella term

All MLP's are NN's, but not all NN's are MLP's

MLP is a type of ANN with multiple layers

weights :- control the influence of inputs b/t neurons ;

bias :- Constants added to weighted input, allowing for adjustment of activation thresholds & shifting the o/p, for more flexibility

Neuron :-

= Neuron''

Say ex :- Age = 43 → $y = 0.042x - 1.53$ | $z = \frac{1}{1+e^{-y}}$ → 0.57 = o/p

Linear eq' | activation func' (Φ) | Binary C/f

say $y = 0.042(x) - 1.53$

$y = 0.042(x_1) + 0.008(x_2) + 0.2(x_3) - 1.53$

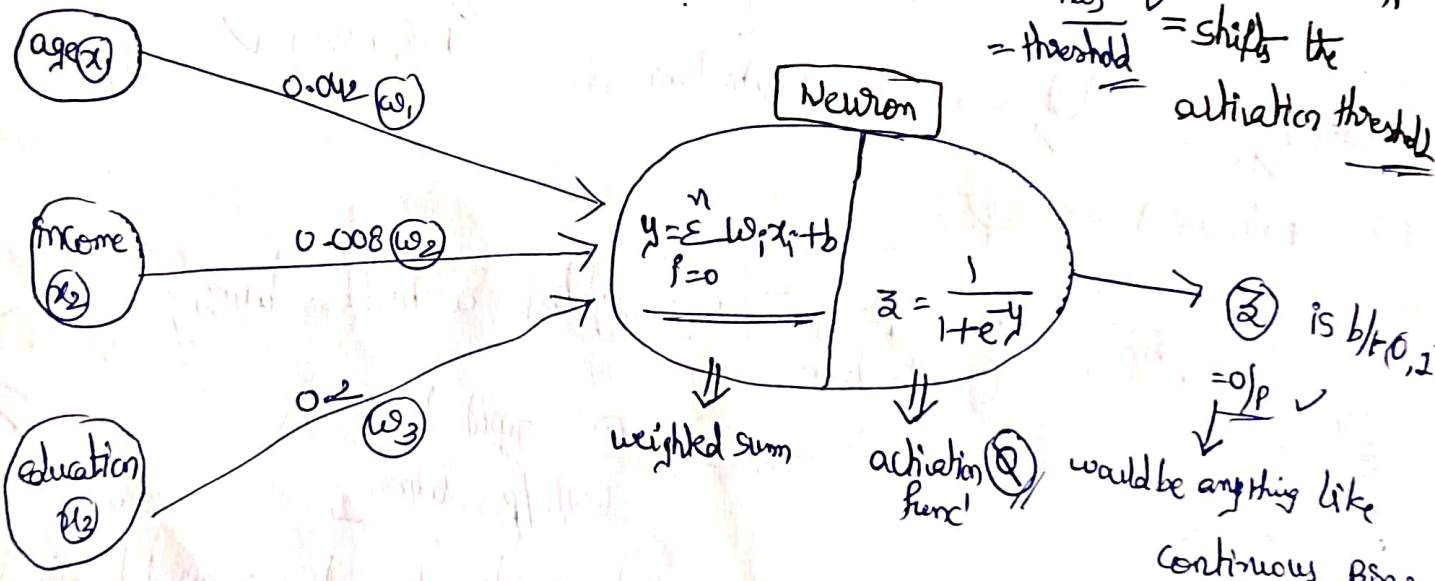
x, x_1, x_2, x_3 = feature like age, income, educ...

$y = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$

$$y = \sum_{i=0}^{n} w_i x_i + b$$

$w_1, w_2 \ldots$ = weights ✓ = Importance of I/p

b = bias ✓ = threshold = shifts the activation threshold



① neuron :- means node (or) unit, is the basic computational unit of a NN'; that mimics the behavior of a biological neurons;

✱ Activation func' decides, if the neuron "fires" & passes info' forward ✓

~~ANN~~	(Biological) Natural NN (say)
Neuron	→ Neuron
dendrites	→ capture info from other neuron ✓
axon	→ passes the info. ✓ (like passage)
2nd neuron	→ o/p

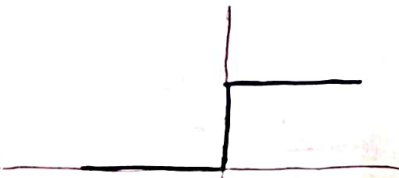
⊛ Perceptron [Indebik] :-

It is an algorithm for supervised learning of binary classifiers; (T or F)

It can be seen as a single unit of an ANN & also known as prototype for neural nets;

— single layer NN with the unit step func' acts as an activation func';

unit step function :- $g(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise;} \end{cases}$



graph

prediction = $\hat{y} = g(f(x))$

$= g(w^T x + b)$

↑ indicates "applying activation func";

w = weights

b = bias ; = threshold

perceptron update Rule :-

for each training sample x_i :-

(i) Initialize weights ✓

(ii) $w = w + \Delta w$

$b = b + \Delta b$

where $\Delta w = \alpha (y_i - \hat{y}_i) \cdot x_i$

$\Delta b = \alpha (y_i - \hat{y}_i)$

where α = learning rate = $[0, 1]$

actual y	prediction ✓ $\hat{y}$	$y - \hat{y}$
1	1	0
1	0	1
0	0	0
0	1	−1

perceptron simply :-

- A line in 2D which creates two regions to classify the two classes;
- 3D — plane makes ✓

Code :- import numpy as np

class perceptron:

def __init__

⊛ disadv :- perceptron not works on non linear data;

⊛ playground.tensorflow.org ✓

⊛ MLP :- ① weights & bias
② How to denote weights & bias ✓

MLP notations :- data = {m×n}   m = rows
say → n = 4 = columns

	1	2	3	4	y
i	·	·	·	·	

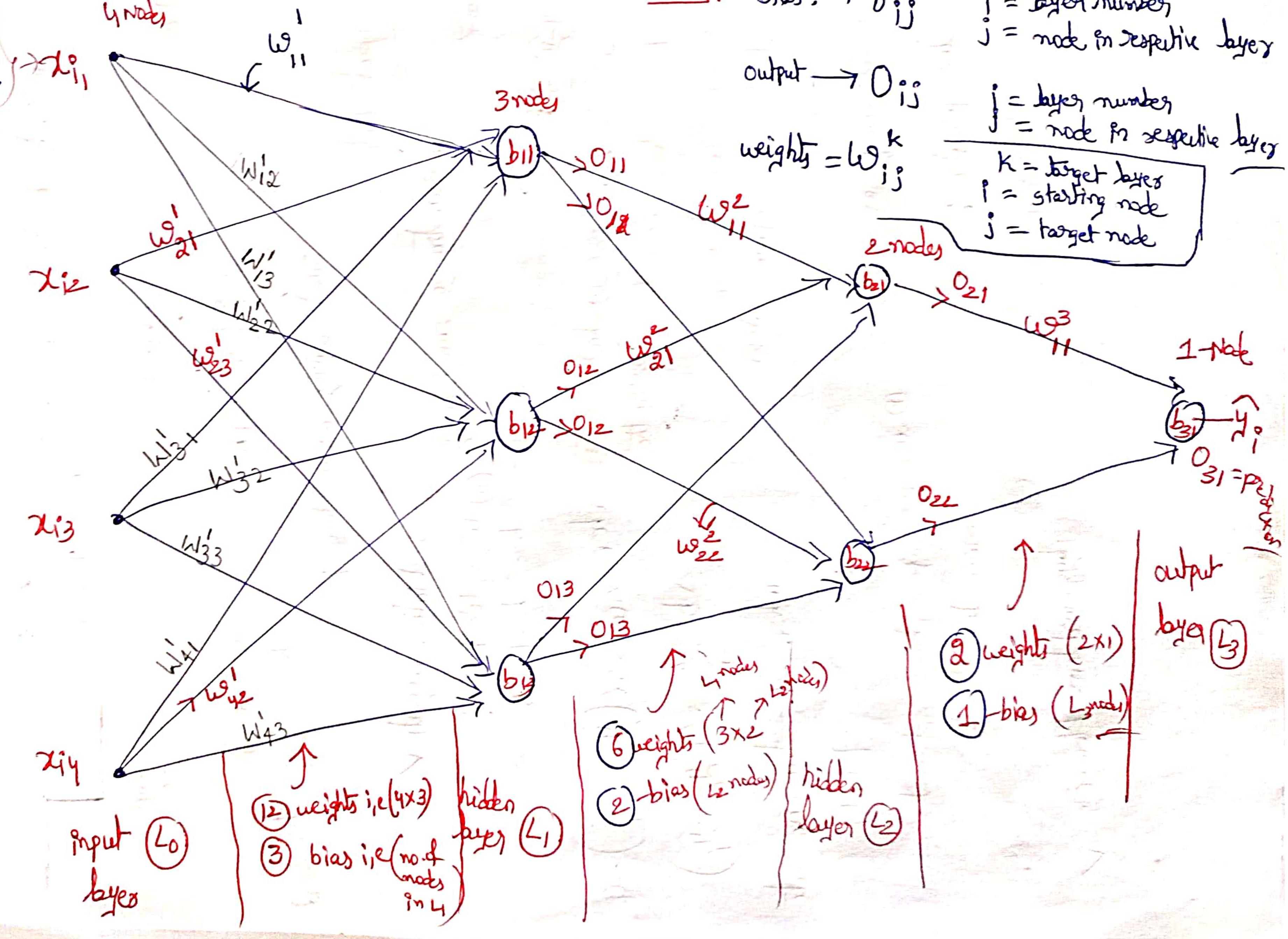
Trainable parameters (w, b) = ? for Backpropagation;

Total weights = 12 + 6 + 2 = 20
bias = 3 + 2 + 1 = 6
} ㉖ Trainable parameters

Notations :- bias :→ b_{ij} i = layer number, j = node in respective layer

output → O_{ij} i = layer number, j = node in respective layer

weights = W^{k}_{ij} k = target layer, i = starting node, j = target node



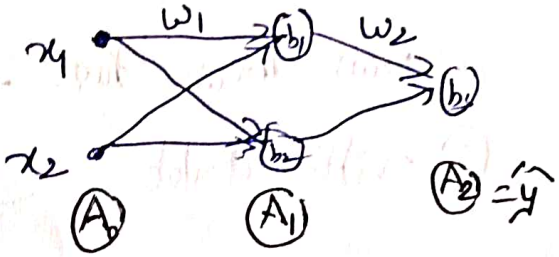
Activation Functions (AF) in DL :-

It is very important component in NN's; introduces (Non-linearity)

— **Definition :-** In ANN, each neuron forms a weighted sum of its input & passes the resulting scalar value through a func' refd to as an activation function (or) transfer func'. If a neuron has (n) inputs then the output (or) activation of neuron is

$$a = g(w_1x_1 + w_2x_2 + \cdots + b)$$ where (g) ref'as (AF);

— In short, (AF) if a mathematical Gate ✓ b/t input & output;



$$z_1 = w_1 A_0 + b_1$$

$$A_1 = g(z_1)$$

↓ AF' ✓

if AF' = linear then $A_1 = z_1$

$$A_2 = g(w_2 A_1 + b_2)$$

$$= w_2 A_1 + b_2$$

$$= w_2(w_1 A_0 + b_1) + b_2$$

$$= w_2 w_1 A_0 + w_2 b_1 + b_2$$

$$= w' A_0 + b'$$

$A^2 = w' A_0 + b'$

↓ output ↓ input

$\hat{y}$ ~ its relation is linear ✓

Ideal AF' Quality :-

(i) should be non-linear

(2) should be Differentiable (to help gradient disent BP')

(3) " be computationally inexpensible;

(4) zero centered; & normalized ✓

↓ (mean = zero)

Ex:- tanh ✓

(5) Non-saturating (means should sieze the input data range)

↓

Ex:- ReLu

↓

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

Ex: (1) Sigmoid → o/p [0, 1]

(2) tanh → o/p (-1, 1)

Ex: ↓

Saturating AF'

↓

Causes vanishing gradient problem

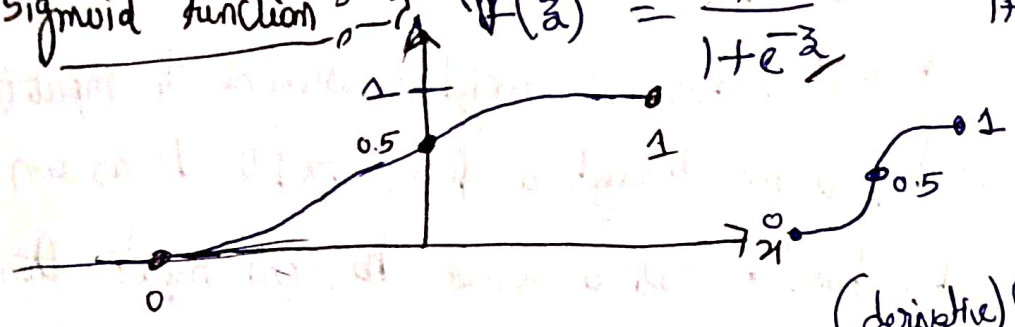
① Sigmoid function :- $\sigma(\vec{z}) = \frac{1}{1+e^{-\vec{z}}}$ ✓

z = weighted sum of all inputs ✓

if $z=0 \Rightarrow \sigma(z) = 0.5$

$z = \max \Rightarrow \sigma(z) = \approx 1$

$z = \min \Rightarrow \sigma(z) = 0$



(derivative) $\sigma'(z) = 0$

adv :- ① o/p always like in b/t $[0,1]$ → feels like probability @ o/p layers ✓ ∴ use for Binary classification problems ✓ (0 = No, 1 = Yes)

② Non-linear func' → Capture non-linear data;

③ Differentiable → helped in Backpropagation (BP) i.e $\frac{\partial L}{\partial w}$ ✓

Disadv :- ① saturating function → because of its range; it causes vanishing gradient problem ✓ means BP: $W_n = W_o - \eta \frac{\partial L}{\partial w}$; if $\frac{\partial L}{\partial w} = 0$ then $W_n = W_o$ ✓ means No update will take place

② Non-zero Centered; → that's why we don't use in hidden layers; ✓ → used only in o/p ✓

→ Causes training slow ✓

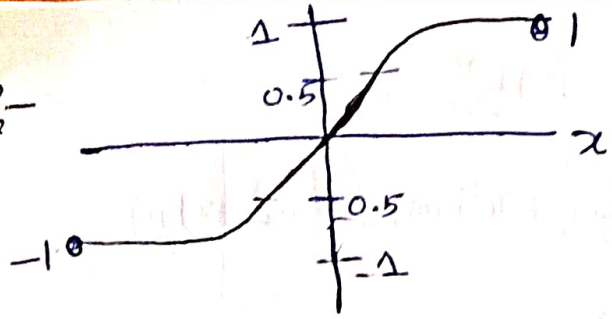
→ means normalized the values you may get either +ve or -ve but not means = 0 ✓

③ Computationally expensive; because of ⓔ component i.e $\sigma(z) = \frac{1}{1+e^{-z}}$ ✓

hidden layers → ReLU

o/p layer → sigmoid → Binary classification;

(2) Tanh :-



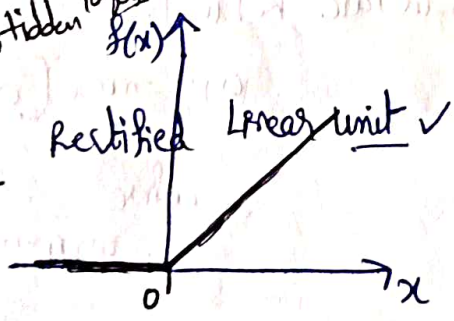
(i) $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \Rightarrow f'(x) = 1 - \tanh^2(x)$

(ii) range $(-1, 1)$

Adv :- (i) Non linear
(ii) Differentiable
(iii) Zero-centered; → ⊕ve, → ⊖ve } Training faster than sigmoid ✓

Disadv :- (i) saturating func' —prone to→ vanishing gradient prob'
(ii) computationally expensive;

(3) ReLu :- (best ✓ for Hidden layers)



mostly used;

sigmoid is used @ o/p layers because of its output range (0, 1) that helps in binary classification ✓

$f(x) = \max(0, x)$ ← weighted sum

means for ⊖ve values $f(x) = 0$
for ⊕ve values $f(x) = x$

Adv :- (i) Non linear; because of its $\max(0, x)$
(ii) Non-saturated in ⊕ve region
(iii) Computationally inexpensive
(iv) converge, faster

disadv :- (i) not completely differentiable;
(ii) not zero centered; to overcome this we will use Batch normalization ✓

* Best choice for hidden layers is (ReLU);

* Big disadv :- is dying ReLU problem; (dead ReLU)

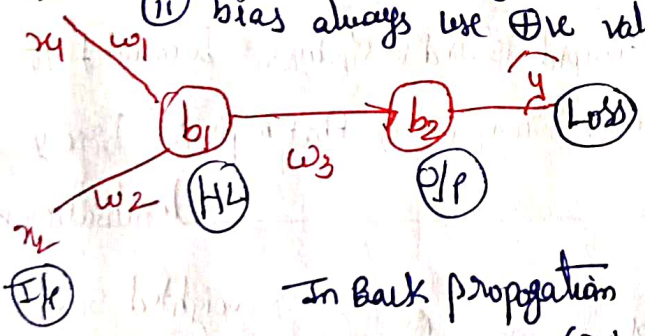
Causes :- (1) large negative bias
(2) most inputs being negative;

Explanation

the issue arises because the derivative of ReLU is (zero) for negative inputs; if a neuron's input consistently falls into the negative range due to its weight & bias, it will always o/p zero & the zero gradient prevents the neurons from learning & updating its weights ✓

Solution to this :-

(i) set low learning rate (if updates not happing then node said to be dead node)
(ii) bias always use +ve value i.e (0.01) ✓ (iii) use its variants ✓



@HL we have ReLU like

$$a_1 = \max(0, z_1)$$

where

$$z_1 = w_1 x_1 + w_2 x_2 + \cdots + b_1 < 0$$

then $a_1 = 0$

then $\frac{\partial a_1}{\partial z_1} = 0$ then w_1, w_2 can't updated;

then node @HL called as (dead) node;

In Back propagation

$$w_1 = w_1 - \eta\left(\frac{\partial L}{\partial w_1}\right)$$

$$w_2 = w_2 - \eta\left(\frac{\partial L}{\partial w_2}\right)$$

L = Loss

* Lets take regression problem;

Loss depends on $\hat{y}$ $\rightarrow$ w_3, b_2, a_1

$a_1 \rightarrow z_1$ $\rightarrow$ w_1, w_2, x_1, x_2, b_1

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$$ (Chain Rule)

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_1} \frac{\partial a_1}{\partial z_1} \frac{\partial z_1}{\partial w_2}$$

if it is zero then finally $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2} = 0$

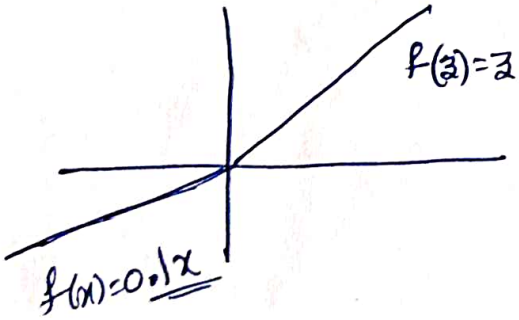
if $z_1 = w_1 x_1 + w_2 x_2 + b_1 < 0$

$z_1 = -ve$ when ① High learning rate ② ~~this is 0~~ ① High -ve bias ✓

—once a neuron dead, it never re-work;

ReLu variants :- 2 types < Linear < Leaky ReLu, parametric ReLu
Non-linear < ELu, Selu

Leaky Relu :- $f(z) = \max(0.01z, z)$ if $z > 0 = z$
$z < 0 = \frac{1}{100} z$ = fraction of z ✓



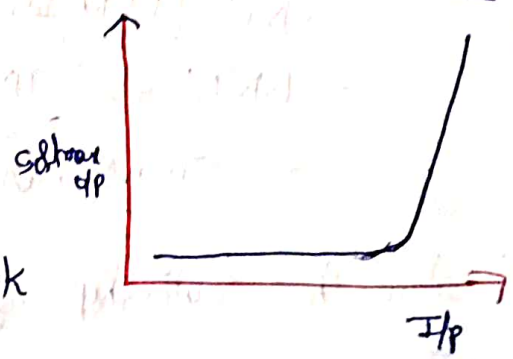
⊛ Neural networks learn on based on connections; Forward & Backward propagation

⊛ ReLu → Regression Linear unit → step func'
Sigmoid → classification Non-" " → ReLu, sigmoid, Tanh
Softmax → multi-class ✓ exponential Linear unit → softmax, softplus;

⊛ Softmax function :-
- designed to handle multi-class classification problems;
- It transforms raw o/p scores from a NN into probabilities;
  i.e vector of raw scores (logits) Ex:- p(x) = 0.5, p(y) = 0.3 p(z) = 0.2
- It works by squashing the output values of each class into range 0 to 1 while ensuring that sum of all probabilities equals (1) ✓

⊛ Given a vector of Real numbers $\vec{z} = (z_1, z_2, \ldots, z_k)$
then softmax function computes:

$$\sigma(\vec{z}) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, 2, \ldots, K$$



z_i = input score
K = Total no of classes
$\sigma(\vec{z})$ = probability of class (i)

Ex:- Say $\vec{z} = [2, 1, 0.1]$

Ans:- (i) $e^{\vec{z}} = [e^2, e^1, e^{0.1}] = [7.39, 2.72, 1.11]$

(ii) $\Sigma = 7.39 + 2.72 + 1.11 = 11.22$ ✓

(iii) $\text{softmax}(\vec{z}) = \left[\frac{7.39}{11.22}, \frac{2.72}{11.22}, \frac{1.11}{11.22}\right] = [0.66, 0.24, 0.1]$
66% 24% 10%

⊛ Typically combined with cross-entropy loss for training;
↓
loss func' used for classification purpose ✓

Softmax Vs Sigmoid

Feature	Softmax	Sigmoid
output type	probability distribution (sum to 1)	Indep. probabilities
used for	multi-class classification	Binary classification
formula	Involves all outputs	Each output computed indep';

adv:-

① Differentiable & suitable for gradient based learning

② Ideal for tasks like Image classification, Text ,, } ***

③ used in output layers of classifiers like
- CNN (MNIST, CIFAR-10)
- RNN, LSTM's
- Transformers (Token classification)

④ - show the relationship graph b/t sigmoid & softmax ✓

Code for Softmax

```
import numpy as np
def softmax(x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum(axis=0)
logits = np.array([2.1, 1.0, 0.1])
probs = softmax(logits)
print(probs)  # [0.659, 0.242, 0.09...]
```

Softplus :- smooth approximation of ReLU function, it allows for differentiability @ all points unlike ReLU which is not differentiable @ 0;

(ii) $f(x) = \log(1+e^x)$ — o/p always +ve

range : $(0, \infty)$ — function is smooth & differentiable everywhere

Intuition :-

* for large +ve x, e^x becomes very large ; so: $f(x) = \log(e^x) = x$ ✓

* " " -ve x, e^x approaches zero so: $f(x) = \log(1) = 0$

So simply it behaves like:

- ReLU for large positive (x)
- Near zero for large negative (x)
- smooth curve in between ✓

Comparision :-

Input	ReLU	Softplus
-2	0	0.13
0	0	0.69
2	2	2.13

So, softplus always gives a small +ve even for negative inputs ✓

* $\frac{d}{dx}[\text{softplus}] = \text{sigmoid}$; [derivative of softplus = sigmoid] ✓

* Adv :-

- Smooth version of ReLU
- Differentiable everywhere ; - better for optimization ;
- Avoids "dying ReLU problem"

* Disadv :-

- Computationally expensive than ReLU ✓
- In pratice, ReLU outperforms it in terms of speed & training behavior ✓

Gradient Descent :-

Gradient descent is one of the most popular algorithms to perform optimization by far the most common way to optimise NN's;

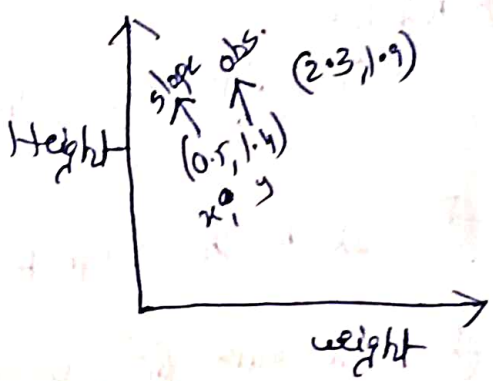
$J(\theta)$ = loss func' ⊛ it min' the loss or loss func' ✓

— In ML & DS, DL we optimise a lot of stuff;

— Linear Regression :- Say : Height = $mx + c$

= slope × weight + (Intercept)

predicted Height = intercept ⊕ (Slope × weight)



— We'll start by using GD to find the intercept; once we understand how GD works, we'll use it to solve for the intercept & the slope;

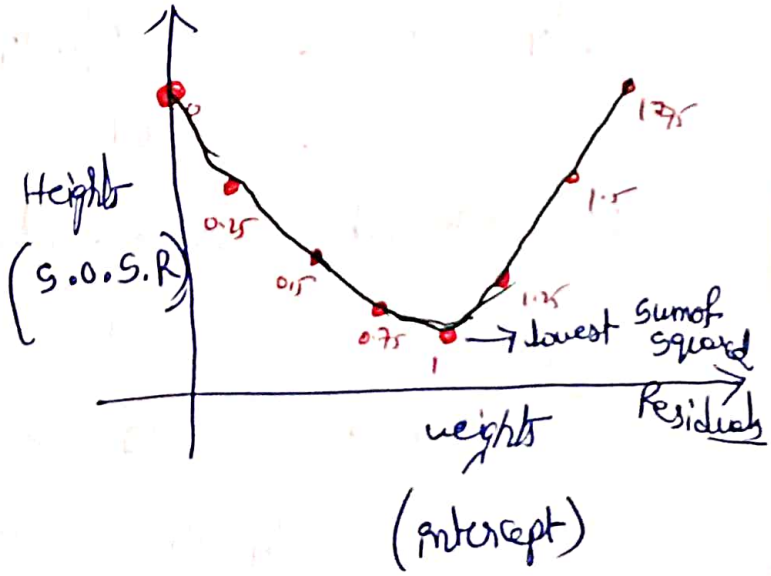
— Say slope = 0.64 ⊛ then just estimate least squares;

— sum of squared Residuals = type of loss func'; (say $1.1^2 + 0.4^2 + 1.3^2 = 3.1$)

when intercept = 0

⊛ what is the best value for intercept is b/t (0.75, 1, 1.25) = ?

Ⓐ solution is :- GD only does a few calculations far from a optimal solution;



Sum of squared residuals = (observed − predicted)^2

$$= \left(1.4 - (\text{intercept} + 0.64 \times 0.5)\right)^2 + \left(1.9 - (I + 0.64 \times 2.3)\right)^2 + \left(3.2 - (I + 0.64 \times 2.9)\right)^2$$

$$\frac{d(\text{S.O.S.R})}{d(\text{intercept})} = -2\left[1.4 - (I + 0.64 \times 0.5)\right] - 2\left[1.9 - (I + 0.64 \times 2.3)\right] - 2\left[3.2 - (I + 0.64 \times 2.9)\right]$$

the derivate, GD will use it to find where the S.O.S.R is (min) or lowest;

* It is where the slope of curve = 0 then that gives best intercept; (Take baby steps to get optimal)
* this makes (GD) very useful when it is not possible to solve for where the derivative = 0, & this is why (GD) can be used in so many diff cases;

— put (I = 0) in the derivate eq' :- we get $\frac{d(\text{SOSR})}{dI} = -5.7$

— step size = -5.7×0.1 → learning rate ✓

= −0.57

— new intercept = old intercept − step size

= 0 − (−0.57) = (0.57) ✓

2nd cycle :- ~~step size = 0~~ New I = 0.57 sub. in derivate eq' & get (−2.3)

step size = −0.23

New intercept = 0.57 − (0.23)

= 0.57 + 0.23 = (0.8) ✓

Now $I(0.8) \Rightarrow -0.9$

step size $= -0.9 \times 0.1 = -0.09$

New intercept $= 0.89$ ✓

like next " " $= 0.92$ ✓

" " $= 0.94$ ✓

" " $= 0.95$ ✓

— Now each step gets smaller & smaller the closer, we get to the bottom of the curve;

(Note):- the least squares estimate for the intercept is 0.95 ✓

(*) GD :- It is an optimization algorithm used to train NN' by updating their weights & biases to min' loss func'; [Backbone of NN']

Why do we need GD :-

(i) We predict o/p using weights & biases;

(ii) The loss func' tells us how wrong the prediction is;

(iii) Goal :- Adjust the weights/biases so that the loss becomes as small as possible;

visual diagram :-

Input → weights ⊕ Biases → Activation → o/p

↓

Compute loss

↓

Back propagate error

↓

Compute Gradients

↓

Update weights using (GD)

↓

Repeat for several epochs;

How it works :-

① Initialize weights & biases randomly;

② Feed forward the I/p data through the network to get predictions;

③ Cal' the loss (error) b/t prediction & actual o/p;

④ Back propagate the error to compute gradients;

⑤ Update the weights in the direction opposite to the gradient;

$$w \leftarrow w - \eta \left(\frac{\partial L}{\partial w}\right)$$

η → LR

$\frac{\partial L}{\partial w}$ → gradient of loss

equation

Challenges :-

- Local minima / saddle points
- Learning Rate :- Too high - overshoot
  Too slow - slow learning ;
- Vanishing / exploding gradient ;

✱ prediction ($\hat{y}$) $= wx$

True value : y_{true}

(MSE) loss function : $(y - y_{true})^2$

- we compute the gradient :-

$$\frac{dL}{dw} = 2 \times [y - y_{true}]$$

then update :-

$$w = w - \eta \left(\frac{dL}{dw}\right) \checkmark$$

Ex :- climbing down a Hill to find the treasure ;

say :- you are on top of a big hill, blind folded & someone tells you : - "the treasure is at the bottom of hill" But you can't see anything, so how do you find the treasure ?

- steps :- (i) you feel which direction the hill is going down,
  (ii) you take a small step downhill ;
  (iii) again, you feel the slope & take another step down ;
  (iv) you keep doing this until ...
  (v) you reach the bottom - where the treasure is ↓ ✓

Hill = error (or) loss
Treasure = min. error
Each step = weight update
Feeling slope = cal 'gradient'
walking slowly = learning rate (step size)

⊛ Weight Initialization Techniques :-

⊛ Why weight initialization is imp :- ① Initialize the parameters (w, b)

② Choose an optimization algorithm

— out of all the points, ① is most Imp;

③ Repeat these steps :-

- Forward propagate an input
- Compute the cost func' (loss func')
- Compute the gradients & cost w.r.t. params using BP;
- Update each parameter using the gradients a/c to optimization algorithm.

(like GD steps)

— Vanishing Gradient problem
Exploding " "
Slow converge "

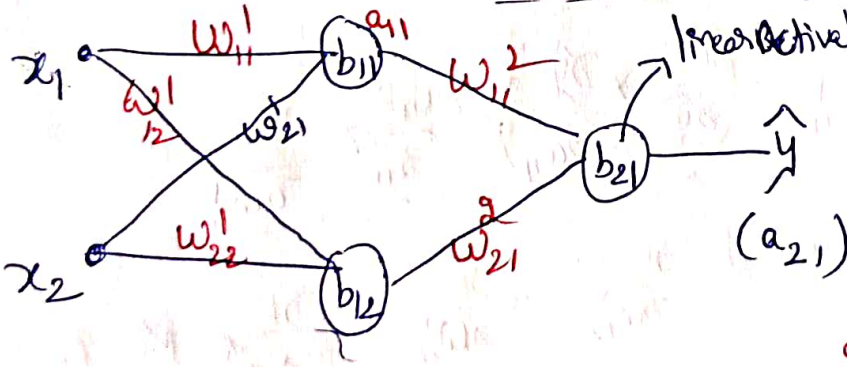
because of above issues to

— Overcome, WIT came;

⊛ Wrong WI Techniques :- (what not to do)

— Case ① :- Zero Initialization :- (means all weights zero)

Regression Example :- student package How much based on CGPA



⊛ Test with ReLU
Tanh
Sigmoid

Let us assume here, $w = 0$, $b = 0$ then

$z_{11} = 0$
$z_{12} = 0$

then

$a_{11} = 0 = a_{12}$ equal to zero

Say @ b_{11}, b_{12} Put ReLU then

$a_{11} = \max(0, z_{11})$

where $z_{11} = w^1_{11} x_1 + w^1_{21} x_2 + b_{11}$

$a_{12} = \max(0, z_{12})$

where

$z_{12} = x_2 w^1_{22} + x_1 w^1_{12} + b_{12}$

⊛ BP $w_{11}^1 = w_{11}^1 - \eta\left(\frac{\partial L}{\partial w_{11}^1}\right)$

$\downarrow$ 0 because of a_{11}

$w_{11}^1 = w_{11}^1$ ⟹ means (no) update happing ✓

— No training will take place;

similarly with Tanh :- $a_{11} = \frac{e^{z_{11}} - e^{-z_{11}}}{e^{z_{11}} + e^{-z_{11}}} = \frac{1-1}{1+1} = 0$ ✓

like $a_{12} = 0$ then automatically $a_{11} = a_{12} = 0$ ✓

No update happing ✓

Try with

⊛ Sigmoid :- put @ HL¹ :- $a_{11} = \sigma(z_{11}) = 0.5$

$a_{12} = 0.5$ } means $a_{11} = a_{12}$

$w = w - \eta\left(\frac{\partial L}{\partial w}\right)$ → cal' for all (4) weights

like

$$\frac{\partial L}{\partial w_{11}^1} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial a_{11}} \times \frac{\partial a_{11}}{\partial z_{11}} \times \frac{\partial z_{11}}{\partial w_{11}^1} = x_1$$

likewise do $\frac{\partial L}{\partial w_{12}^1}$ & $\frac{\partial L}{\partial w_{22}^1}$ & $\frac{\partial L}{\partial w_{21}^1}$ ✓

$$\frac{\partial L}{\partial w_{12}^1} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial a_{12}} \times \frac{\partial a_{12}}{\partial z_{12}} \times \frac{\partial z_{12}}{\partial w_{12}^1} = x_1$$ ✓

$$\frac{\partial L}{\partial w_{21}^1} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial a_{11}} \times \frac{\partial a_{11}}{\partial z_{11}} \times \frac{\partial z_{11}}{\partial w_{21}^1} = x_2$$ ✓

$$\frac{\partial L}{\partial w_{22}^1} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial a_{12}} \times \frac{\partial a_{12}}{\partial z_{12}} \times \frac{\partial z_{12}}{\partial w_{22}^1} = x_2$$ ✓

say $a_{11} = a_{12}$, $z_{11} = z_{12}$ } So, because of that, (all initialized value (z_{ij}), sigmoid@HL)

even 1000 neurons in (HL) also it acts as a (single) node; (make Linear model ✓)

— In short, Non-linearity not existing in (HL);

— Its behavior like a perceptron;

Case ②:- Non-zero Constant Value:- $w = 0.5$, $b = 0.5$

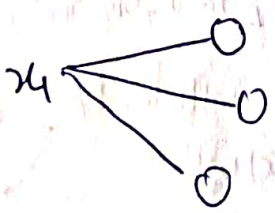
$a_{11} = \max(0, z_{11}) \neq 0$

$z_{11} = w_{11}^1 x_1 + w_{21}^1 x_2 + b_{11} \neq 0$

likewise $a_{12}, z_{12} \neq 0$

but ($z_{11} = z_{12}$, $a_{11} = a_{12}$) because of $w = b$ ✓

$\frac{\partial L}{\partial w_{11}^1}$, $\frac{\partial L}{\partial w_{12}^1}$, $\frac{\partial L}{\partial w_{21}^1}$, $\frac{\partial L}{\partial w_{22}^1}$ Same as above Case;



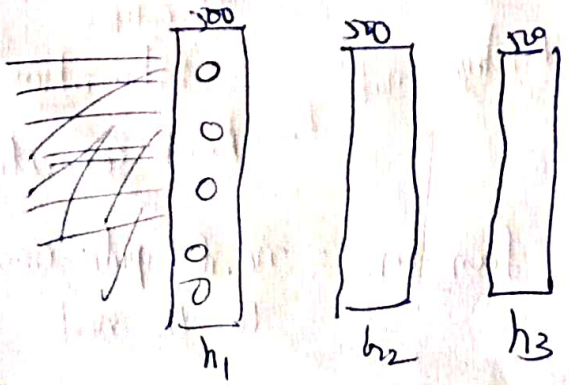
3 nodes acts as Here is single (node)

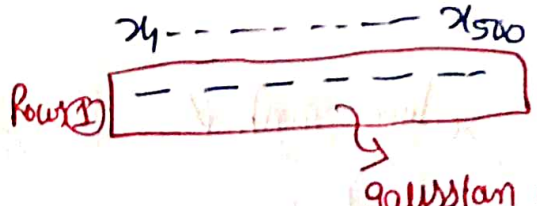
So we have only one option i.e., randomly initial the w, b;

Case ③:- Random Initialization:- cases ① Small Random values — Tanh/Sigmoid

② large R.V's;

Take = Rows = 1000, Column = 500





gaussian distribution

np.random.randn(500, 500) × 0.01

gives small values (b=0)

* activation $= (\Sigma w_i x_i)$ where $x_i = 0$ to 1 means normalised;
(z) → gives small number; weights like 60 small $= 0.007$

say (Tanh) is our AF' ⇒ means Activation functions gives a very close = value; i.e o/p $= 0$; this causes vanishing gradient problems;

— say sigmoid is almost same, it also causes " " ";

— In summary as you used small (b, w) like 0.0007 — (tanh, sigmoid) → vanishing problem

website :- www.deeplearning.ai/ai-notes/initialization

* What can be done :-

— Heuristics (Jugad) → pratical solution

Glorat (1) Xavier Initial [Tanh, sigmoid] < Normal, updated uniform

(2) He initialisation [relu] < Normal, updated uniform

ko, k4, K7, L3, m1, m2, m6, m7, m9, N2, O4, O9, P9, Q4,

Intuition :- Take small weights ↓

np.random.randn(250,250) × 0.01 ✓ ⇒ Range −0.01 to 0.01

for large weights :- " " " " × 1 ✓ ⇒ Range −3 to 3 ✓ ⇓ becomes −250 to 250 ✓ it causes saturation ✓

$z = \Sigma w_i x_i$

250 nodes ✓

— Variance should be $\frac{1}{n}$ ✓ $= \frac{1}{250}$ ✓

n = A hidden layer nodes' ka back connections;

only on Tanh ✓

$SD = \frac{1}{\sqrt{250}}$

Xavier initialization : (Normal) :- $\sqrt{\frac{1}{fan\_in}}$

fan_in = no. of inputs coming to the node

(4) np.random.randn(2,3) * $\sqrt{\frac{1}{dim}}$ ✓

(or)

Some rough work like = $\sqrt{\frac{2}{fan\_in + fan\_out}}$

(5) He normal :- $\sqrt{\frac{2}{fan\_in}}$ → If you are using with ReLU ✓(keras)

uniform version means :- based on uniform distribution;

① Xavier uniform :- [-limit, limit]

where limit = $\sqrt{\frac{6}{fan\_in + fan\_out}}$

② He uniform :- [-limit, limit] where limit = $\sqrt{\frac{6}{fan\_in}}$ ✓