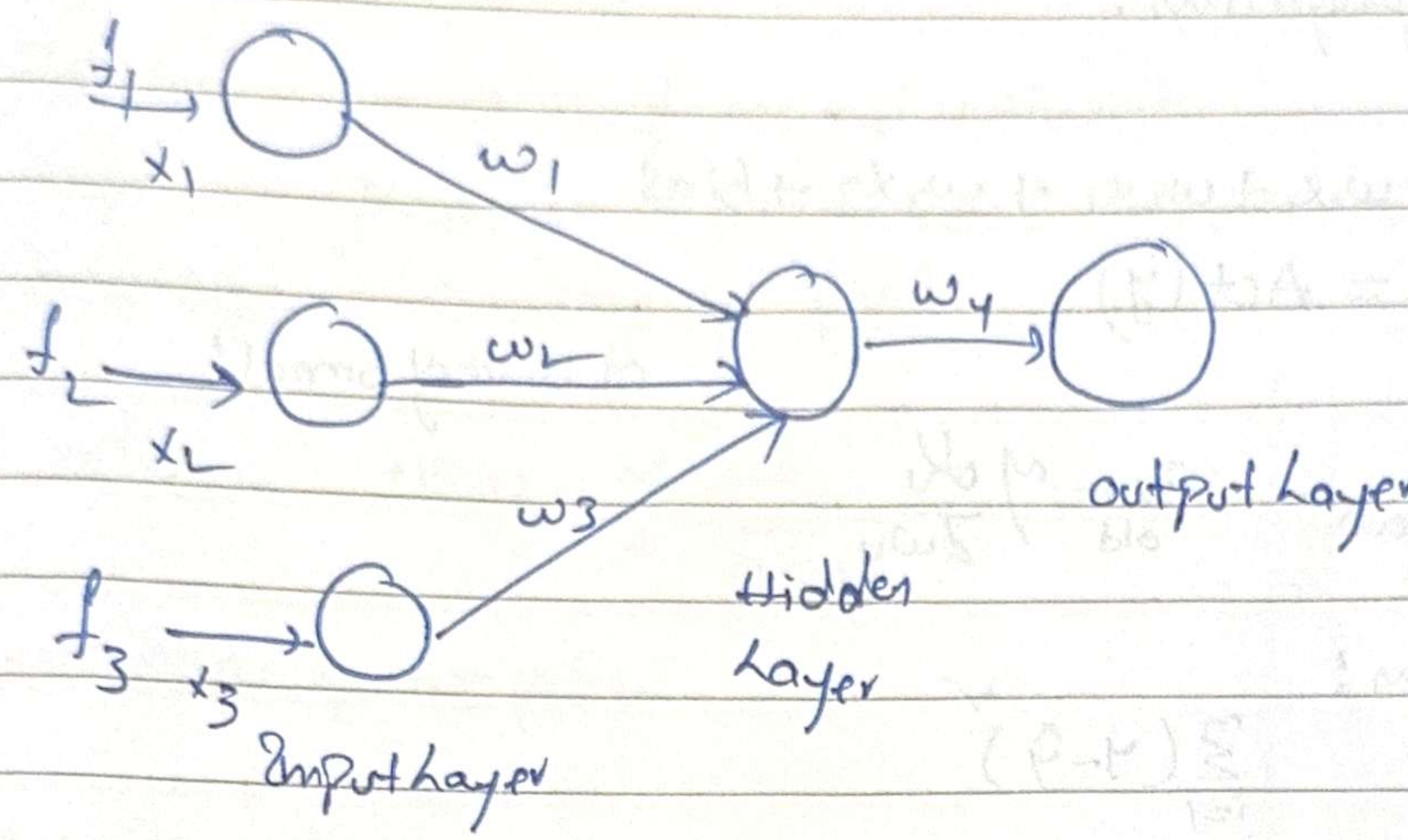
Deep Learning.

ANN
CNN
RNN

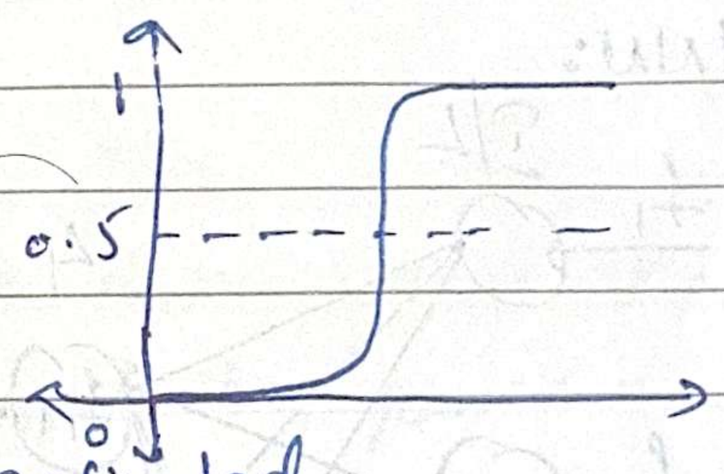
* the first Neural network created is Perceptron



$$y = w_1x_1 + w_2x_2 + w_3x_3 + bias \quad / \quad z = Act(y)$$

Activation functions:

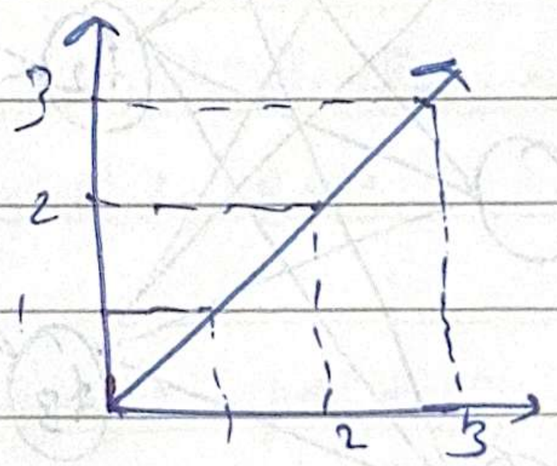
1. Sigmoid Af: $= \frac{1}{1+e^{-y}}$



If the output is '1' then the Neuran is activated else not activated.

2. ReLu Af: $\max(y, 0)$

If y = -ve then its always 0, else



In classification problem it always uses the Sigmoid fn at out Layer if binary.

Multi-class, classification we use Softmax

Loss function:

Backpropagation.

$$y = w_1x_1 + w_2x_2 + w_3x_3 + bias$$

$$z = Act(y)$$

η is very small

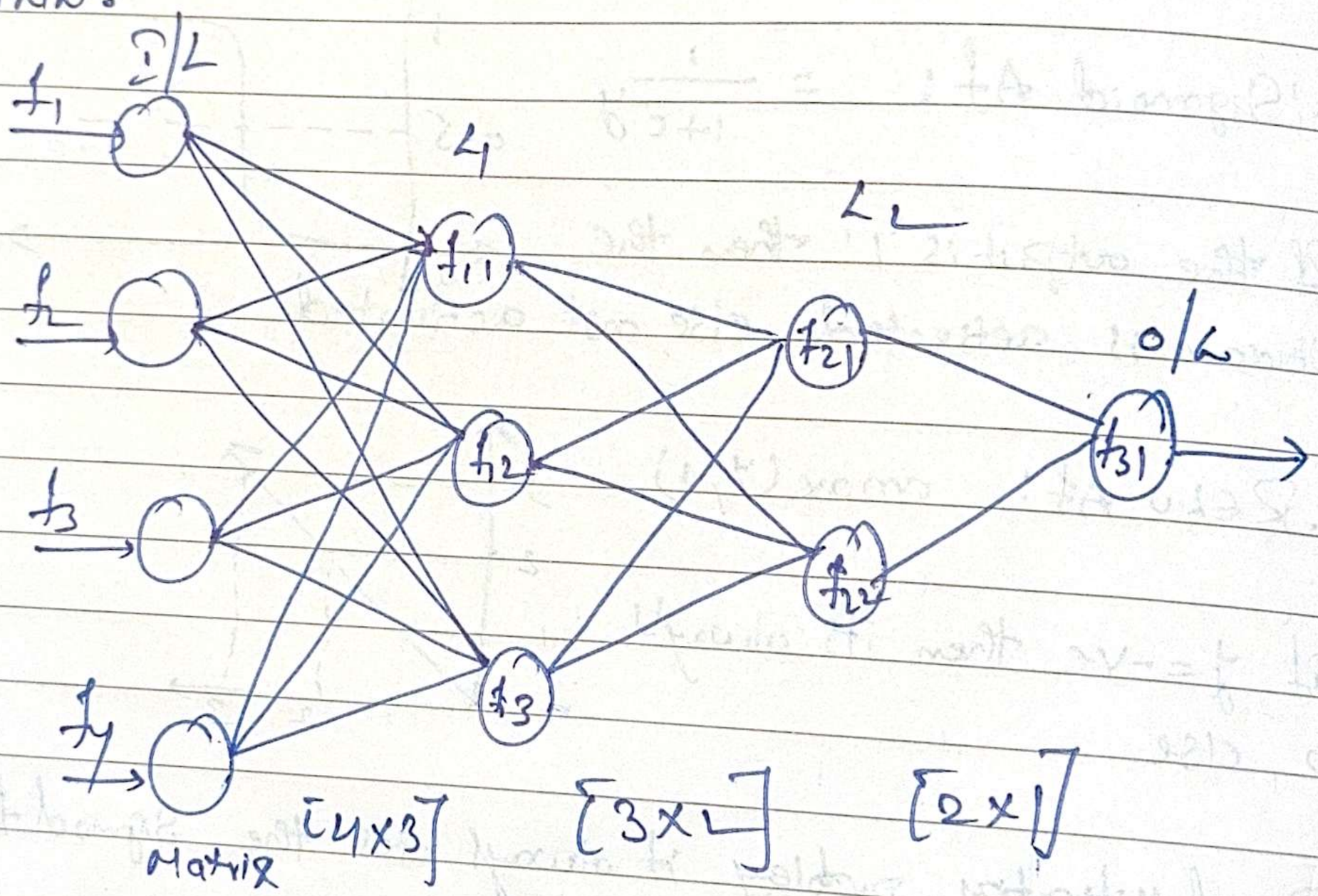
$$w_{4\,new} = w_{4\,old} - \eta \frac{dL}{dw_4}$$

Cost function:

$$\sum_{i=1}^{n} (y - \hat{y})^2$$

In back propagation we find how much the weight is responsible for the the loss by the derivation of the weight.

MNN:



we can select different Act function at different Layes every connection have different weights and every layer have different bais.

Gradient Descent : (optimizers)

Eg: A man on hill

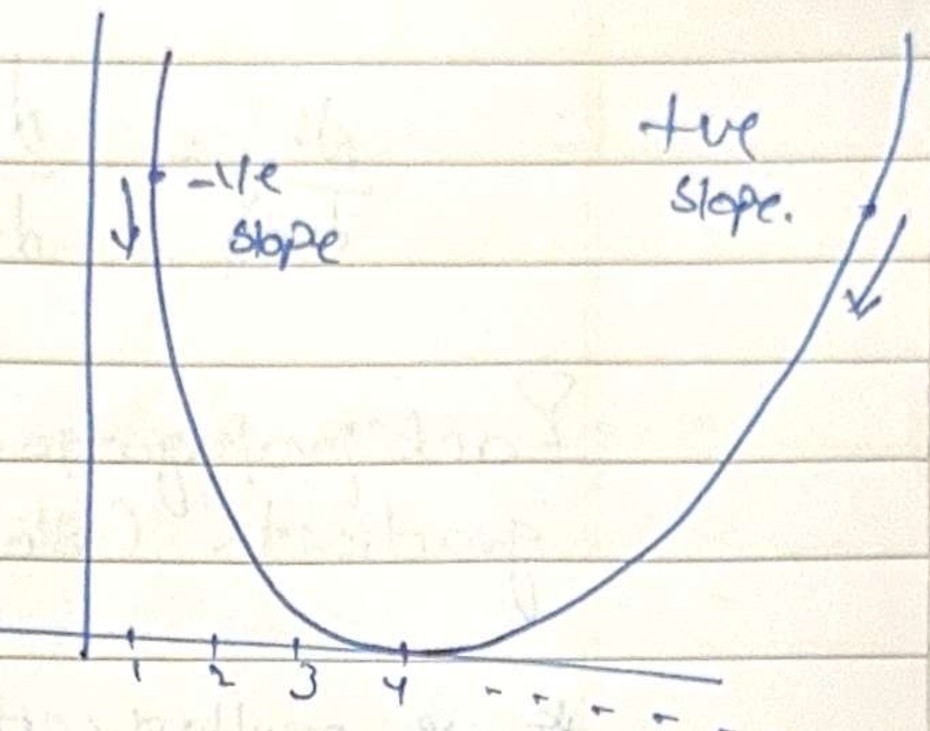
Hill - Loss function

Position - weights of neural network.

Slope — gradient (derivative of Loss wrt weights)

Stepsize - Learning rate (η)

$$w_{new} = w_{old} - \eta \underbrace{\frac{dL}{dw_{old}}}$$



-ve slope ↑ weight

+ve slope ↓ weight.

How fast ↑es or ↓es the weights depends on the η learning rate.

If we take bigger value of η then we may never reach the global min and if we take less η then it may take lot of time and computation power to compute best weights

Batch gradient Descent : uses the whole dataset but very slow for complex data (or) large data sets.

Stochastic Gradient Descent: Speeds up the process by updating weights after one random training sample till all the batches are done.

Mini batch gradient descent: uses small batches like 32 (or) 64 samples : good speed and the stability.

Chain Rule:

2 Hidden Layer Example

Loss function = $(y-\hat{y})^2$

$$\omega_{11}^{new} = \omega_{11}^{old} - \eta \frac{dw}{d\omega_{11}}$$

$$\frac{dL}{d\omega_{11}^1} = \frac{dL}{do_{31}} \times \frac{do_{31}}{d\omega_{22}} \times \frac{do_{22}}{d\omega_{11}^1}$$

Backpropagation uses the chain rule to compute the gradients (Slope)

* we multiply the gradient from the next layer by the derivative of the activation function of the current layer.

* Each Layer passes its error Signal to the previous layer.

* Back propagation computes gradients of the Loss with respect to every weight in the network.

* Gradient tells us How much Small change in that weight changes the Loss. So it measures the propotion of the error caused by the weight.

→ we are multiply the gradients every time So if Slope >1 then grows Exponetialy else Shrinks Exponetialy so gradient value so the first few Layer can Learn by change according to the Loss function.

Vanishing Gradient Problem:

* In early days in deep learning they always used the Sigmoid function as the Act.

⇒ the Sigmoid function converges the values to 0 → 1

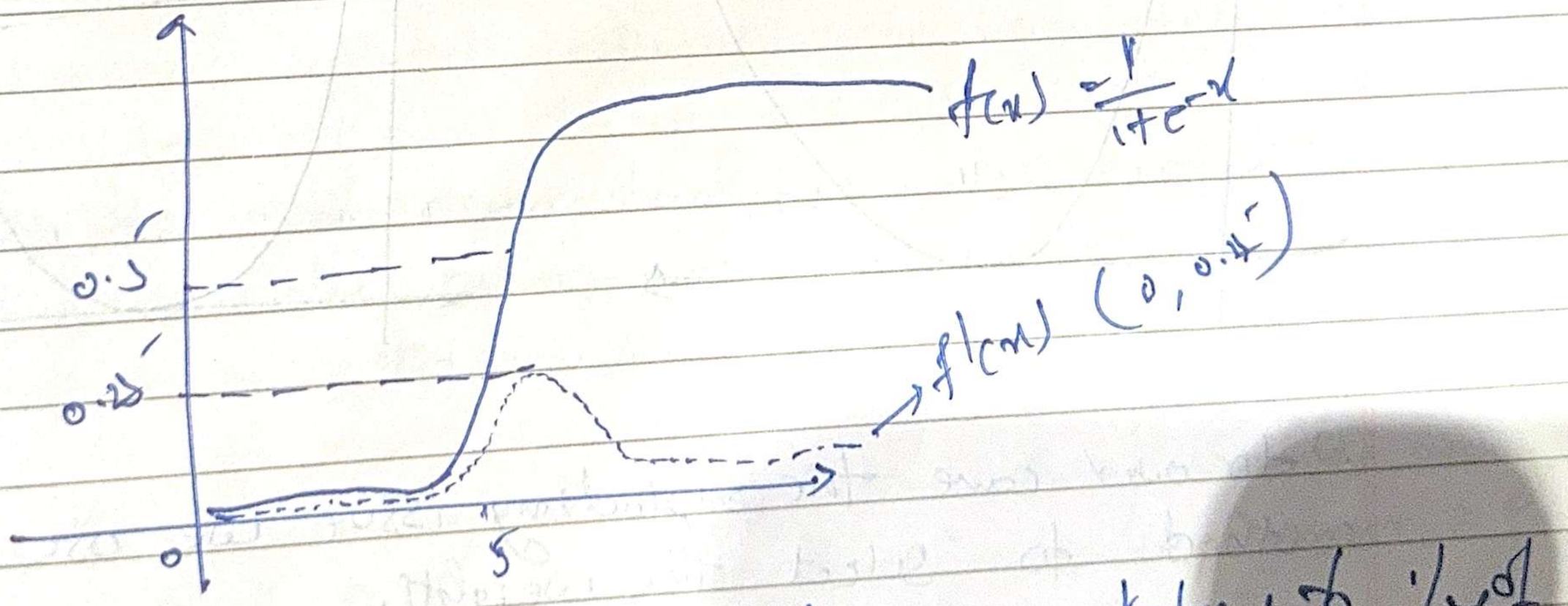
⇒ derivative of the Sigmoid function always lies between 0 to 0.25

* when we have more hidden layers and by chain rule multiply the gradients it tends to 0 there is no change in new weight and old weight.

$$f(x) = \frac{1}{1+e^{-x}} \qquad f'(x) = f(x)(1-f(x))$$

$$f(x) = \tanh(x) \qquad f'(x) = 1 - \tanh^2(x)$$

Sigmoid (0, 1) tanh (-1, 1)



every time we differentiate the value reduced to 1/4 of its original which eventually tends to 0

having deep layered network and using the sigmoid (or) tanh as Act in every layer causes the vanishing problem.

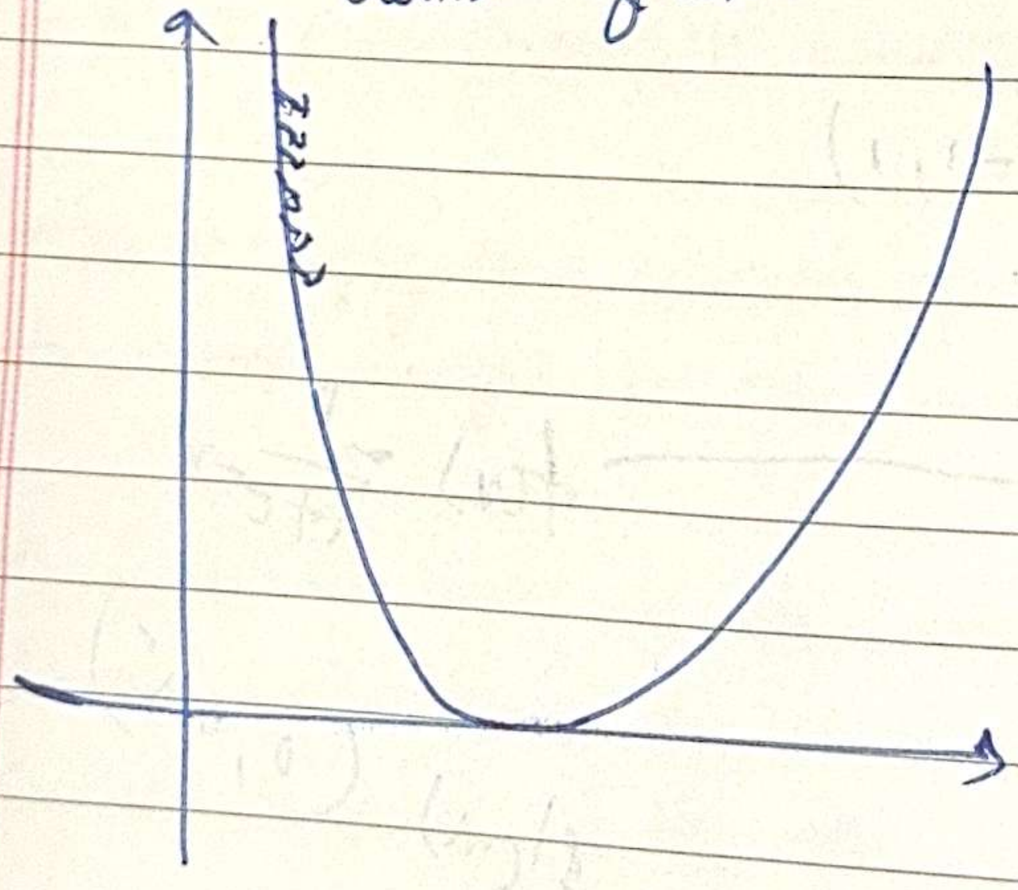
⇒ the layer which are closer input layer the weights wont be changing (or) never converge to global minima are it may take infinite time to reach the global minima.

Gradient Exploding problem:

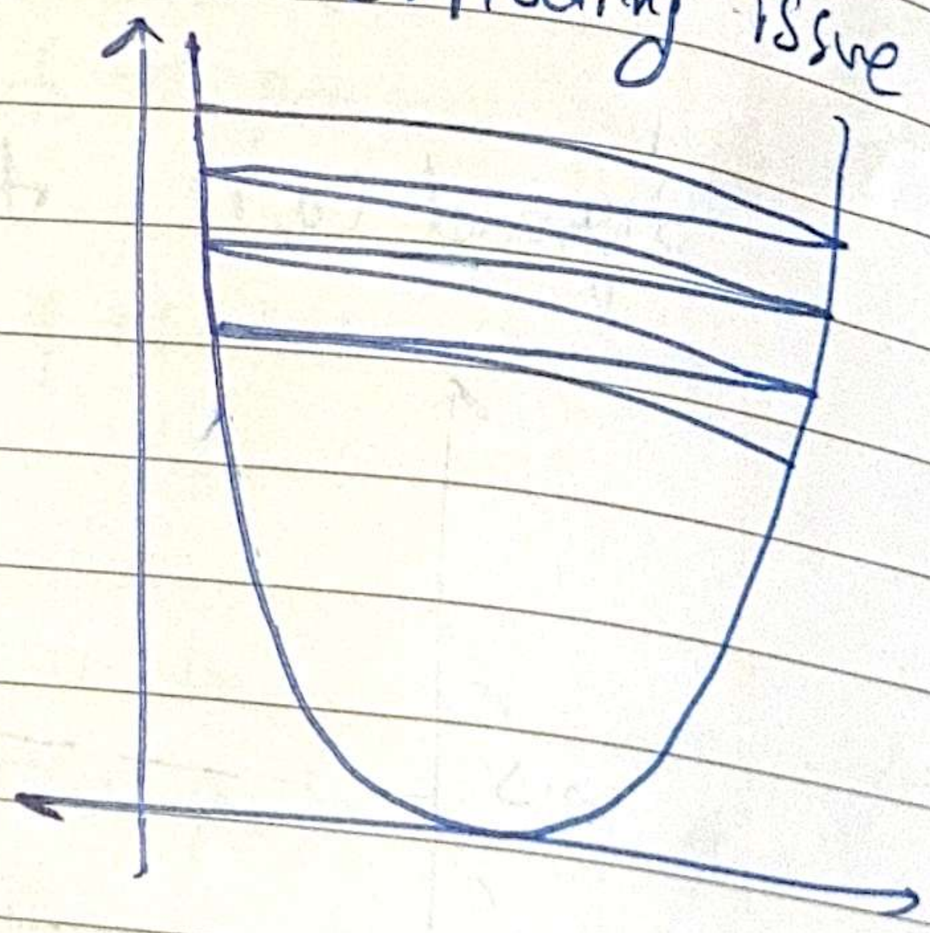
⇒ In vanishing issue the layer closer to input layer can't learn anything because the weight = unew

⇒ In exploding due the larger +ve (or) -ve weights it will never go to the global min it will just zig zag at the top of the hill.

Vanishing issue



Exploding issue



⇒ to over come the exploding issue we use different method to select the weights.

fan in and fan out to calculate the approximate weights which we can assign to the model.

Drop out & Regularization:

Drop out ratio $0 \leq p \leq 1$

⇒ we select the p-value. from (0,1)

⇒ Dropout is the regularization technique to prevent overfitting.

⇒ During the training we randomly drop a fraction of neurons.

→ Droped Neurons will not participate in forward & back ward propagations. for the training step.

⇒ this forces the network not to relay on the few neurons, this will make the system more robust.

In the input layer the p-value (0.1, 0.2) we don't want to lose mor input info, in hidden layers (0.3–0.5)

Each mini batch sees a slightly different network the final Learned network is the average all the networks.

As the No of neurons droped as p-value the total act are the avrage info also drops to maintain we scale up the value by multiplying with $1/(1-p)$

→ This scaling is applied in Every Single training step not just in the final averaging the all mini batches.

to maintain intervence.

Act functions:

Sigmod function:

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

This function is not '0' zero centered only positive

*works good with the binary classification problem and having the very less hidden layers

* $\alpha'(x) = \alpha(x)(1-\alpha(x))$ in every presiding step the gradient value is decreased by 1/4

* This function mostly used at output layer

threshold function:

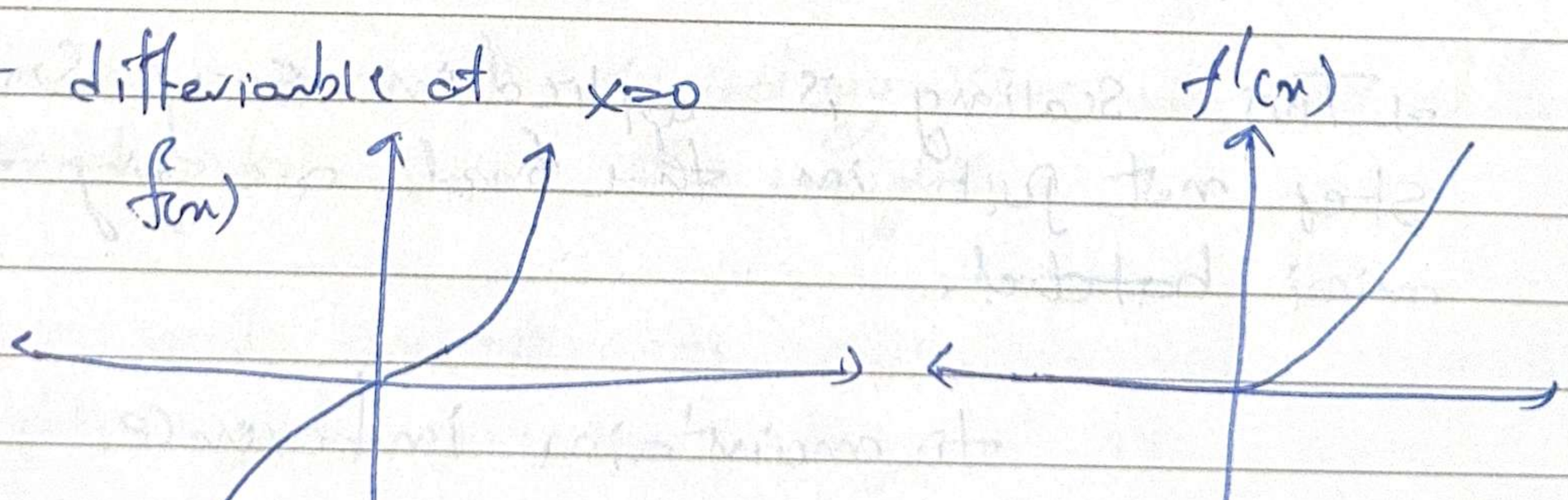
$$\alpha(x) = \tanh x \qquad \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\therefore f'(x) = 1 - \tanh^2 x$$

this is a zero centered function (-1, 1)

if the Act function is not zero centered it never converge to global min the weights zig zag at top of the curve.

Not differiable at x=0



Relu:

$$f(x) = \begin{cases} x & ; x > 0 \\ 0 & : x \leq 0 \end{cases}$$

$f(x) = Max(0, x)$

$$f'(x) = \begin{cases} 1 & ; x \geq 0 \\ 0 & ; x < 0 \end{cases}$$

the Relu is Pieacewise Linear function

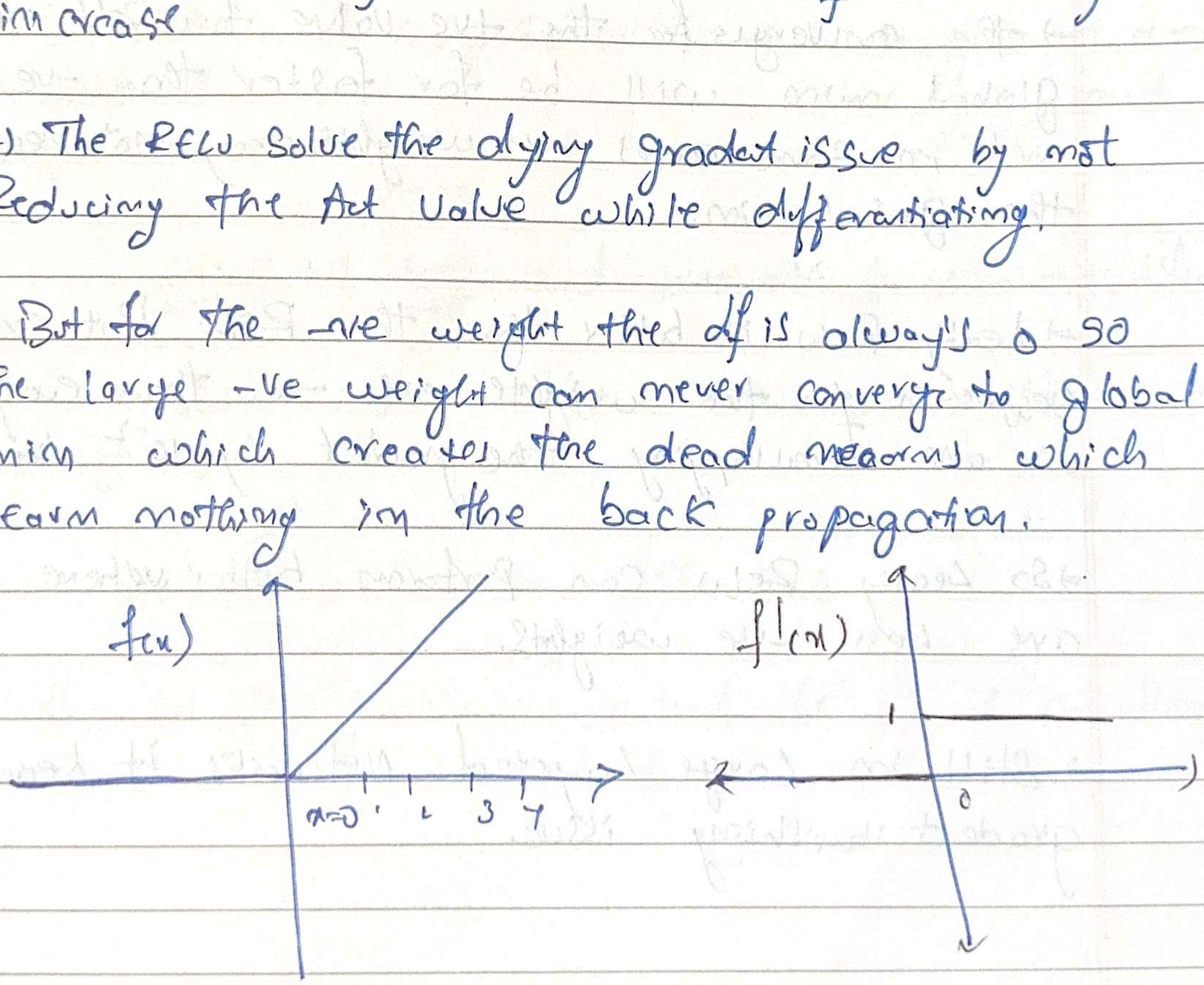
⇒ for the Positive value's the RELU is Linear

⇒ At $x=0$ there is the hinge which makes the RELU Non-Linear

⇒ The non-Linearity ↑'es as the no. of hidden Layer's increase

⇒ The RELU Solve the dying gradent issue by not Reducing the Act Value while differantiating.

⇒ But for the -ve weight the df is alway's 0 so the large -ve weight can never converge to global min which creates the dead neaorns which learn nothing in the back propagation.



Leaky ReLU Act:

⇒ The Leaky ReLU Slove's the dying ReLU issue by adding the 0.01(x) to the negative portions then the derivate will be 0.01 not the zero as in the ReLU.

$$f(x) = \max(0.01x, x)$$

$$f(x) = \begin{cases} 1 ; x > 0 \\ 0.01 ; x \leq 0 \end{cases}$$

⇒ But still we are multiply the gradint with 0.01 this leads to vanishing gradient issue in the -ve weights

→ the converges for the +ve value towards the global min will be far faster than -ve weights and in some cases -ve weights may not reach the global min.

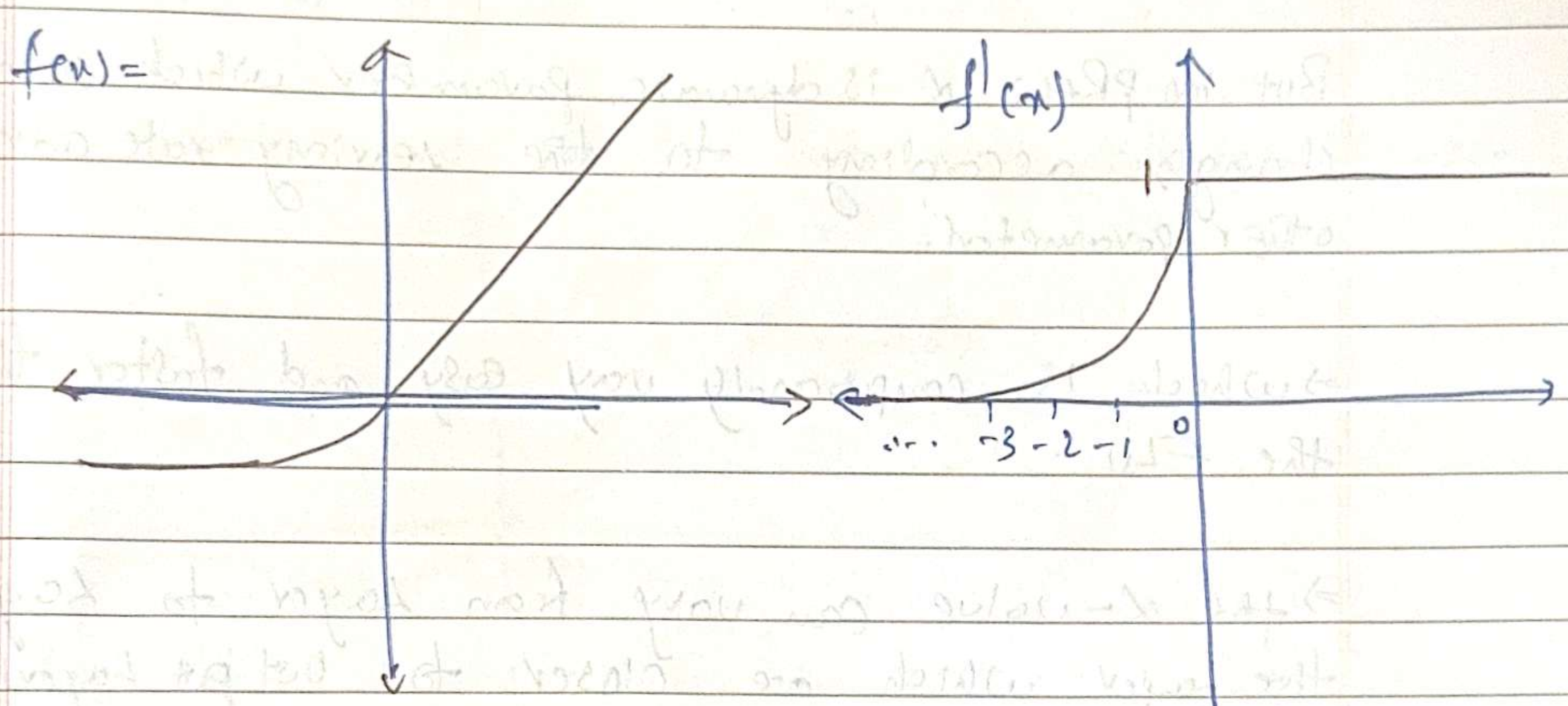
→ Leaky ReLU is better than the ReLU But more sequence of the weights are -ve than every time we are multiplying the gradat by 10^{-2} times.

⇒ So Leaky ReLU can perform better wethere there are less -ve weights.

→ Still in Large Layered Networks it leads to gradent vanishing issue.

Elu (Exponential Linear Unit):

$$f(x) = \begin{cases} x & : x > 0 \\ \alpha(e^x - 1) & ; x \leq 0 \end{cases}$$



α - alpha is hyper parameter.

→ In Relu and Leaky Relu the derivate of x at x=0 is undefined because the slope = 1 when $\lim_{x \to 0^+}$ and slope = 0 when $\lim_{x \to 0^-}$ So the

$$\lim_{x \to 0^+} \neq \lim_{x \to 0^-}$$ and derivate of x=0 not difined

but in some cases it f(x=0) is considered as 0 in the training data

→ we can overcome the issue in the ELU

→ the derivate of the Exponential $(e^x - 1)$ is complex it requires more computational power.

PRelu: (Parametric Relu)

$$f(x) = \begin{cases} \alpha & x > 0 \\ \alpha x & x \leq 0 \end{cases}$$

if $\alpha = 0$ ReLU

if $\alpha = 0.01$ Leaky Relu

But in PRELU α is dynamic parameter which ~~can~~ changes according to the learing rate and other parameters.

⇒ which is computationaly very easy and faster than the ELU.

⇒ the α-value can vary from Layer to Layer the Layer which are closer to Output Layers may have the bigger α and as go deeper towards the input Layer the α is, i.e's so we can overcome the gradeat vanishing issue and the very easy to compute.

Swish:

$f(x) = x * \text{Sigmoid}(x)$ Self gating.

$f(x) = x \cdot \sigma(x)$

$f'(x) = \sigma(x) + x \cdot \sigma(x)(1 - \sigma(x))$

So gradient has two parts:

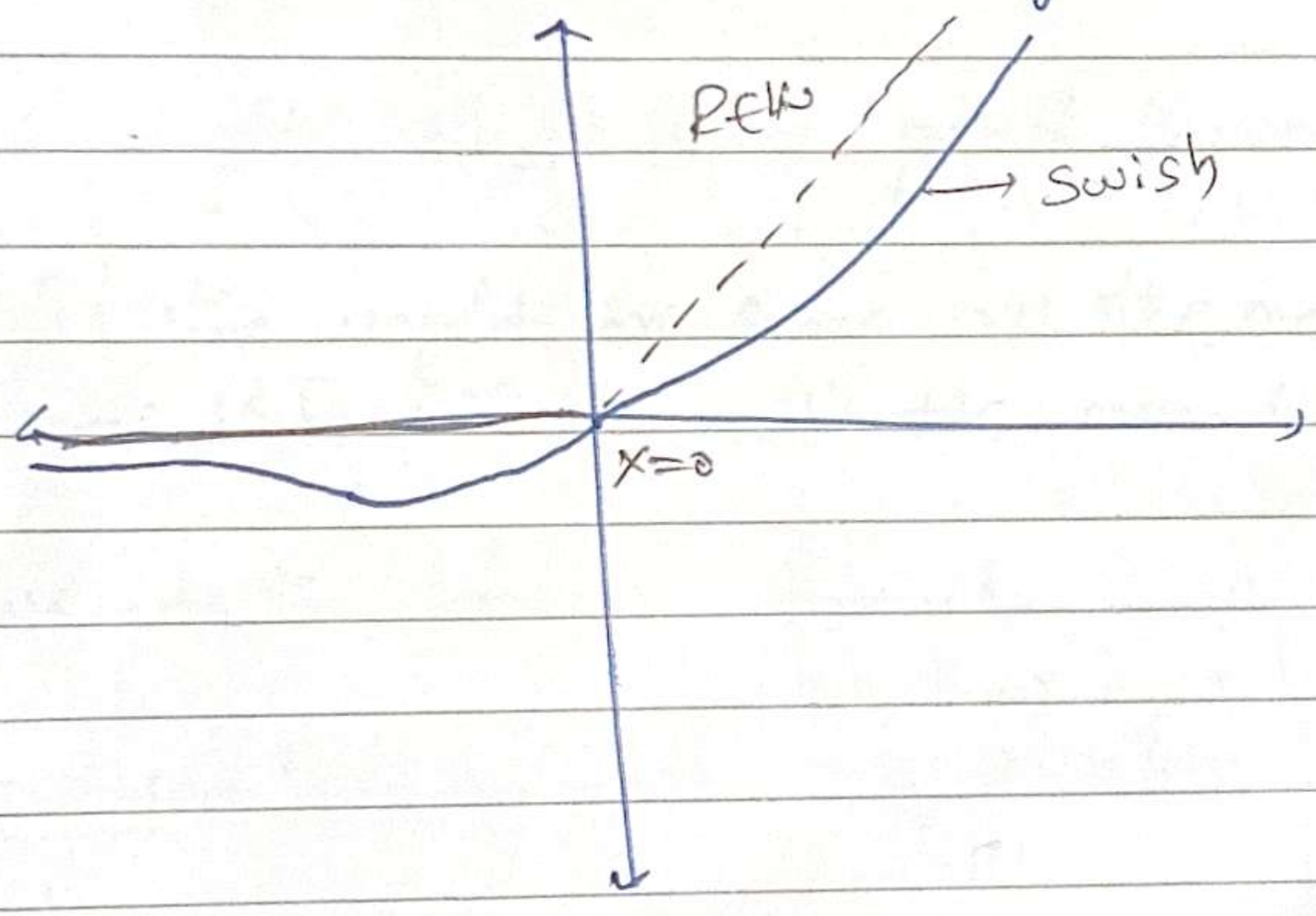
$\sigma(x)$ ensures some gradiant always flows (no dead neurons

$x \cdot \sigma(x)(1 - \sigma(x))$ introduced a self modulation effect.

Sigmoid acts as a Soft gate (between 0, 1)

→ instead of just cutting of negative like ReLU Swish allows some negative values to leak through, but scaled by Sigmoid.

⇒ It better works for the layer ≥ 40 not less than 40



Softplus:

$$f(x) = \ln(1+e^x), \quad f'(x) = \frac{1}{1+e^{-x}}$$

the derivate of the Softplus is the Sigmoid function (0, 1)

weight initialization.

=> when ever we are intializing the weight should not be large we will have the Exploding issue, it too small we will take lot of time to compute.

=> weight should not be same

=> weights Should have good variance.

=> If the weights are same all the neurons recive the same into which kill's the non-linearty.

we use the fan in fanout

Uniform Distribution: $W_{ij} \sim$ uniform $\left[\frac{-1}{\sqrt{fan_{in}}}, \frac{1}{\sqrt{fan_{in}}}\right]$

weight's are selected between the range of (a,b) in uniform.

b. Xaviel/Gorat Distribution:

Xaviel Normal

$W_{ij} \sim N(0, \sigma^2)$

$\sigma = \sqrt{\frac{2}{(fan_{in} + fan_{out})}}$

Xaviel uniform.

$W_{ij} \sim U\left[\frac{-\sqrt{6}}{\sqrt{fan_{in} + fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in} + fan_{out}}}\right]$

3. He init:

He uniform

$W_{ij} \sim U\left[-\sqrt{\frac{6}{fan_{in}}}, \sqrt{\frac{6}{fan_{in}}}\right]$

He Normal

$\sigma = \sqrt{2/fan_{in}}$