Module-2

Training a Neural Network

Training a neural n/w consists of the following basic steps

Step 1: Initialization of Neural N/w

- Initialize weights and bias

Step 2: Forward propagation

- Using the given input 'X', weights 'W' and biases 'b', for every layer we compute a linear combination of inputs and weights and then apply activation function to linear combination
- Finally the final layer, compute $\hat{y}$ by applying appropriate activation function (sigmoid, softmax, linear etc)

Step 3: Compute the loss Function:

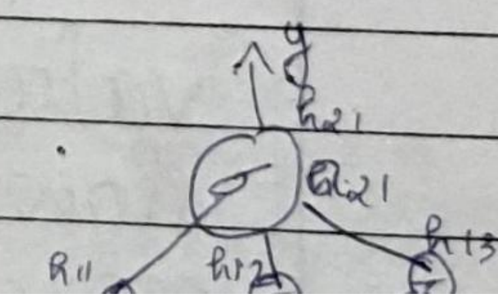
- The loss fn includes both the actual label 'y' and predicted '$\hat{y}$'. calculate using Mean squared error or Cross entropy depends on the pblm we solve.

Step4 : Backward Propagation:

- find the gradients of loss fn with respect to weight, bias etc.
- Using these gradients, we update the values of the parameters from the last layer to the first layer.

Step5 : Repeat 2-4 (steps) for n-epochs till we observe that the loss fn is minimized without overfitting or underfitting the train data

Overfitting : high variance
Underfitting : high bias



Vanishing | Exploding Gradients Problems :-

Vanishing :-

As the backpropagation algorithm advances downwards (or backward) from the olp layer towards the input layer, the gradients often get smaller and smaller and approach zero which eventually leaves the weights of the initial or lower layers nearly unchanged. As a result, the gradient descent never converges to the optimum. This is known as the vanishing gradients problem

Exploding :-

On the contrary, in some cases, the gradients keep on getting larger and larger as the backpropagation algorithm progresses. This, in turn, causes very large weight updates and causes the gradient descent to diverge. This is known as the exploding gradients problem.

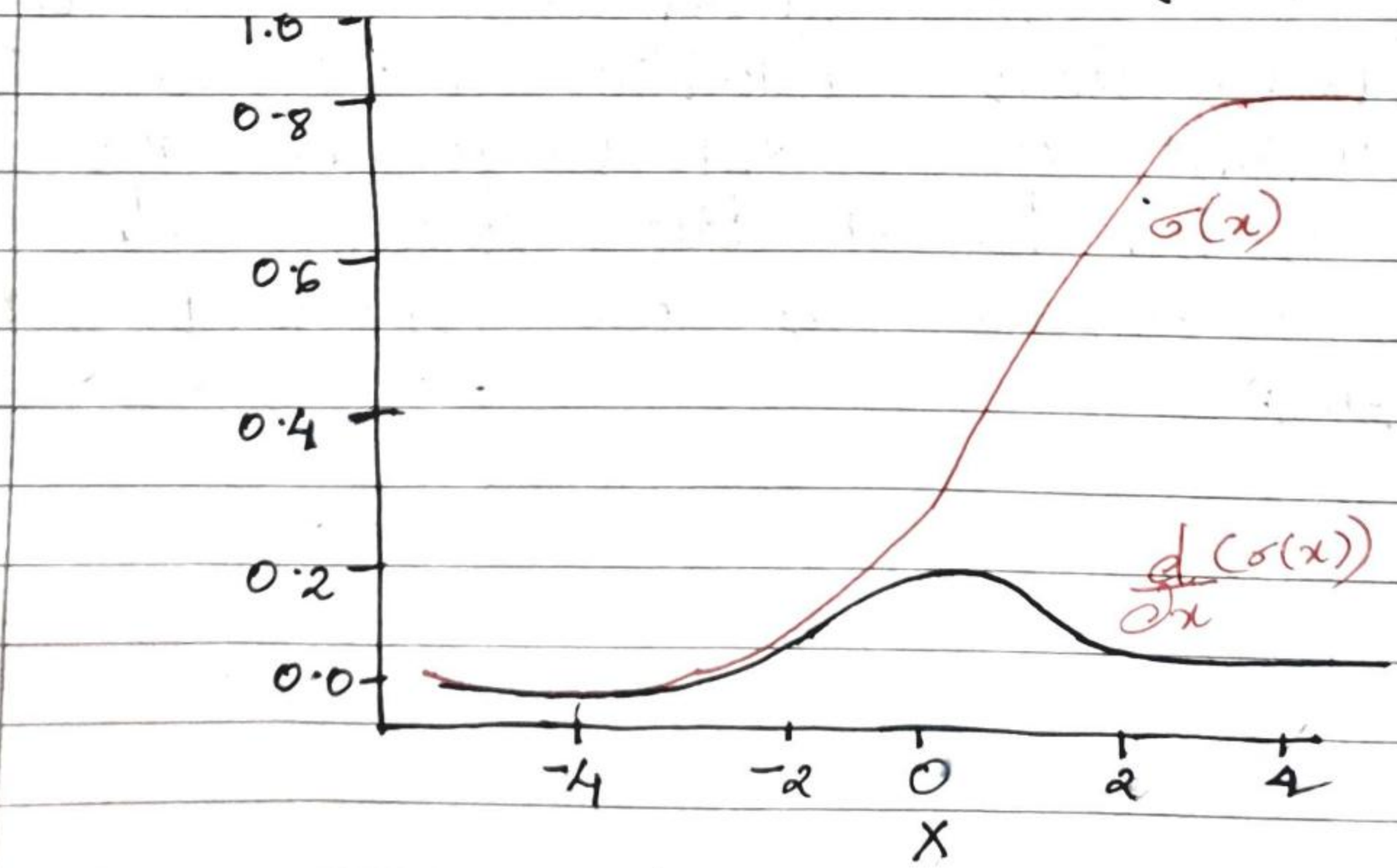
URBAN EDGE

9 Why do the gradients even vanish / explode

* Certain activation functions, like the logistic function (sigmoid), have a very huge difference b/w the variance of their inputs and the outputs

* Wrong activation function and weight initialization techniques may cause the variance of the o/ps of each layer is much greater than the variance of its input.

* Going forward in the n/w the variance keeps increasing after each layer until the activation saturates at the top layer.



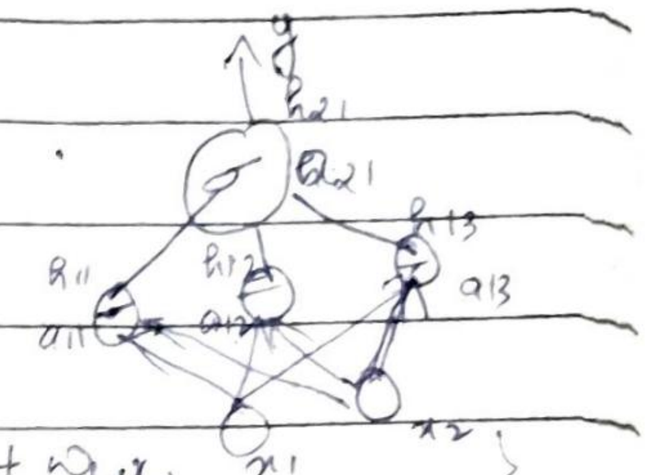
→ Sigmoid function ↑, for large inputs (-ve or +ve), it saturates at 0 or 1 with a derivative very close to zero.

Then, when the backpropagation algorithm works, it virtually has no gradients to propagate backward in the n/w. and whatever little residual gradient exist keeps on diluting as the algorithm progresses down through the top layers. So this leaves nothing for the lowest layers.

* In some cases, the initial weights assigned to the n/w generate some large loss. Now the gradients can accumulate during an update and result in very large gradients which eventually results in large updates to the n/w weights and leads to an unstable n/w.

Overfetting : high variance
Underfetting : high bias

Proper Weight Initialization : eg:



$a_{11} = w_{11}x_1 + w_{12}x_2$
$a_{12} = w_{21}x_1 + w_{22}x_2$
$\therefore a_{11} = a_{12}$
ie $h_{11} = h_{12}$
when $w = 0$ same for all subsequent

Zero Initialization

* If all the weights are initialized with '0', the derivative with respect to loss function is same for every 'w'. Thus all weights have the same value in subsequent iterations. This makes hidden units symmetric and continues for all the 'n' iterations.
ie setting weights to 0 does not make it better than a linear model.

This problem is called as Symmetry Breaking Problem

Random Inits

* * If weights are initialized with a very high values then, $\Sigma wx + b$ becomes significantly higher and leads to exploding gradients pblm

* If weights are initialized with a small values, this leads to the Vanishing gradient pblm

∴ Initializing weights with inappropriate values will lead to divergence or a slow-down in the training of your neural n/w.

* Find Appropriate Initialization values !!! But how ??...

↳ stick on the following rules

1) • The mean of the activations should be zero.

Variance of a model = Variance of the data + Variance of the difference b/w validation error and training error.

(2) • The variance of the activations should stay the same across every layer.

URBAN EDGE

Xavier Initialization:-

Xavier Initialization proposed by Xavier Glorot and Yoshua Bengio is the eveights initialization technique that tries to make the variance of the outputs of a layer to be equal to the variance of its inputs

$$w^{(l)} \sim N\left(\mu=0, \sigma^2=\frac{1}{n^{(l-1)}}\right)$$

$$b^{(l)} = 0$$

& all the weights of layer 'l' are picked randomly from a normal distribution with mean $\mu=0$ and variance $\sigma^2 = \frac{1}{n^{(l-1)}}$, & the no: of

where $n^{(l-1)}$ is the number of neuron in layer $(l-1)$.

- Activation function used in Xavier is TanH.

He Initialization:- (Kaiming Initial)

It is an initialization method for neural n/w that take into account the non-linearity of activation functions such as ReLU Activations.

$$W^{(l)} \sim N\left(\mu=0, \sigma^2=\frac{2}{n^{(l-1)}}\right)$$

(2) Using Non-Saturating Activation Functions :-

* In sigmoid activation function, for large inputs (negative or positive), the output is saturated this is main reason behind the vanishing of gradients. So it is not recommendable to use in hidden layers of the networks.

* Therefore we must use some other non-saturating functions like ReLu and its alternatives.

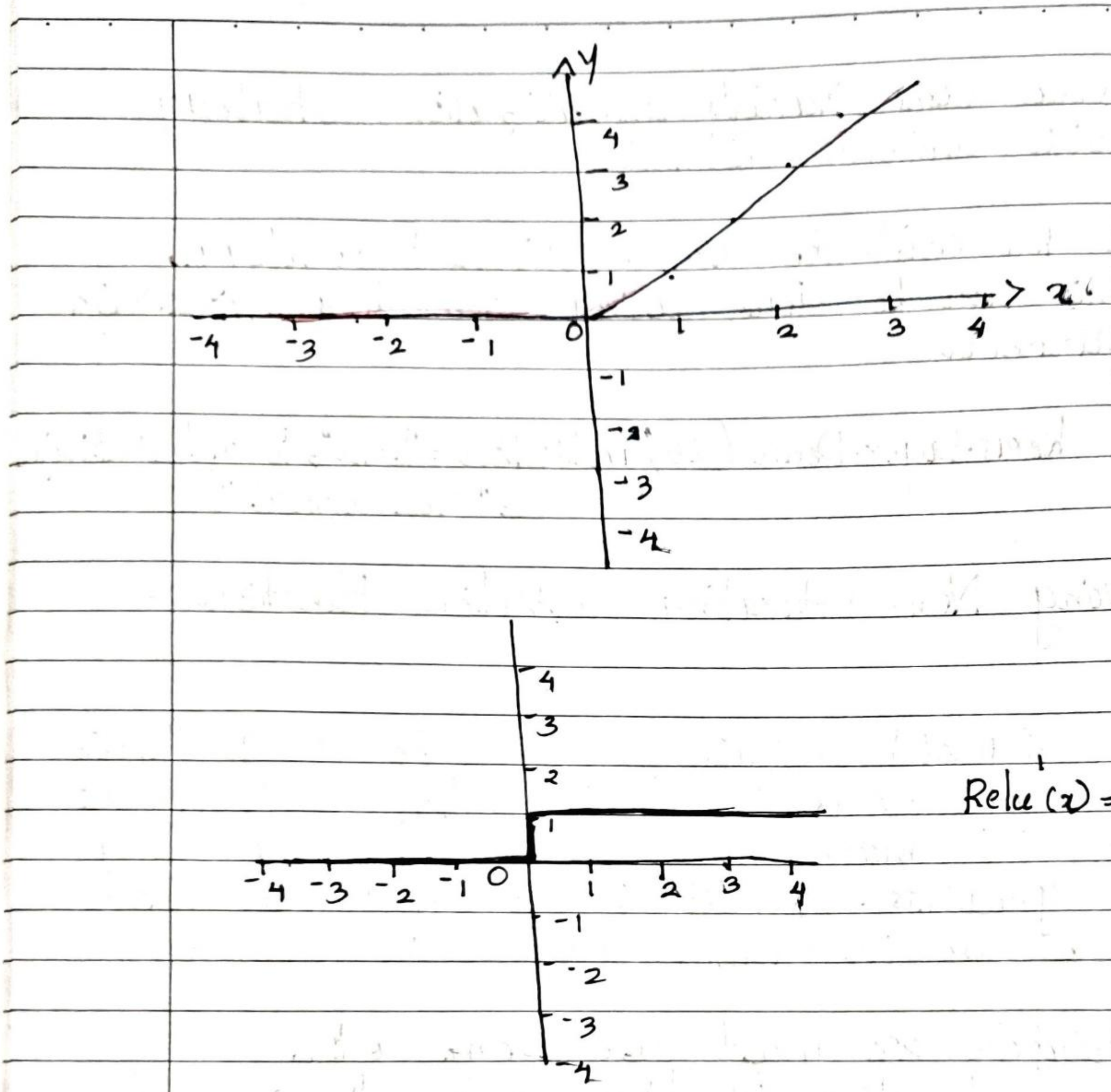
ReLU [Rectified Linear Unit]

Equation for ReLU

$$ReLU(x) = max(0, x)$$

↳ If the Input 'x' is less than 0, set input=0

↳ If the i/p is > 0, set input = input



* we don't get extremely small values. when using a ReLu activation function Instead it gives either 0 or 1. The problem is called 'dead' ReLU problem

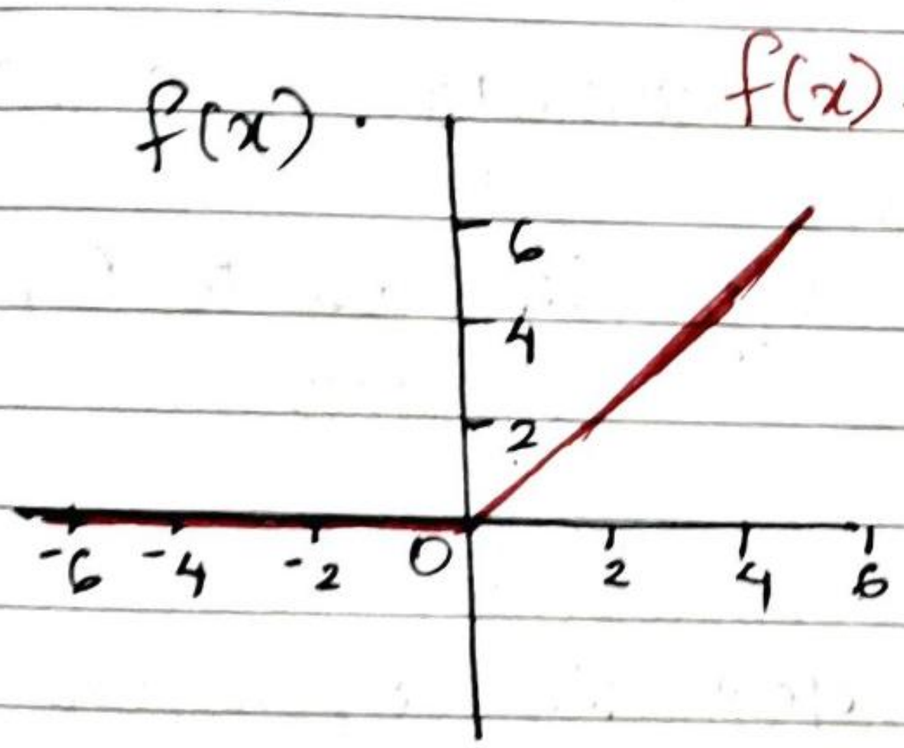
advantage

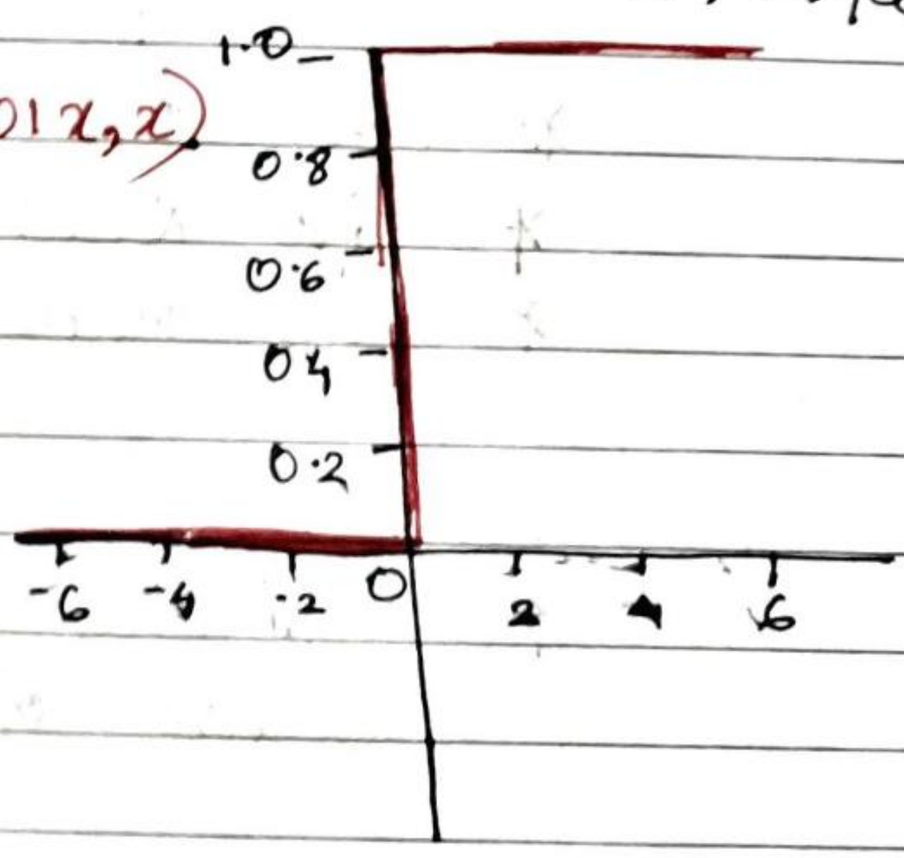
* when the i/p is +ve, there is no gradient saturation problem
* the calculation speed is much faster

* When the i/p is -ve, ReLU is completely inactive, which means that once a negative number is entered, ReLU will die
In forward propagation, it is not a problem but in backward propagation, if you enter a -ve no:, the gradient will be completely zero.

* ReLU function is not a zero-centric function because o/p of the ReLU fn is either 0 or 1

Leaky ReLU function :-





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

$0.01x$

To solve dead ReLU problem

* Set $0.01x$ instead of 0

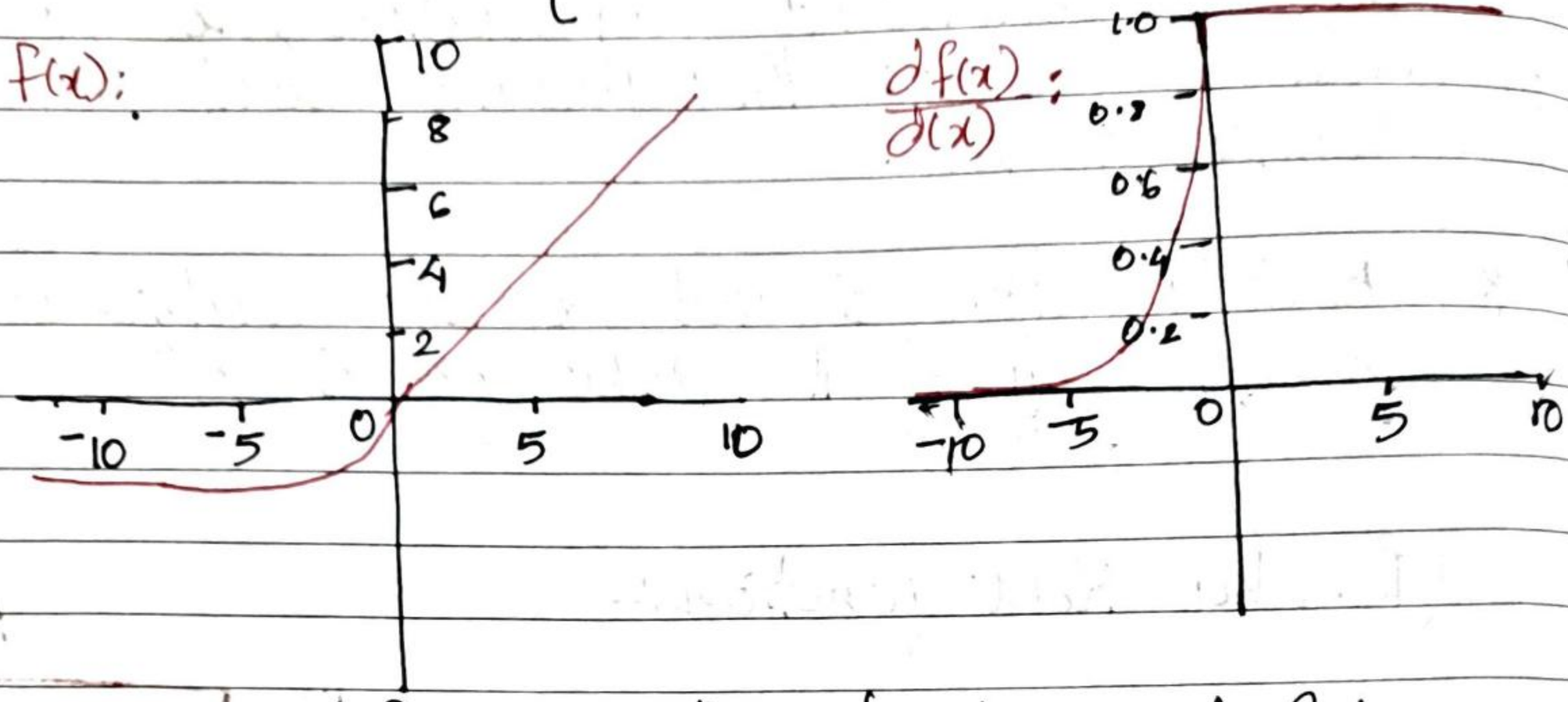
* Avoid dead ReLU problem } advantage
* Faster Computation

* Does not avoid exploding gradient problem } disadvantage.

URBAN EDGE

ELU (Exponential Linear Units) function:-

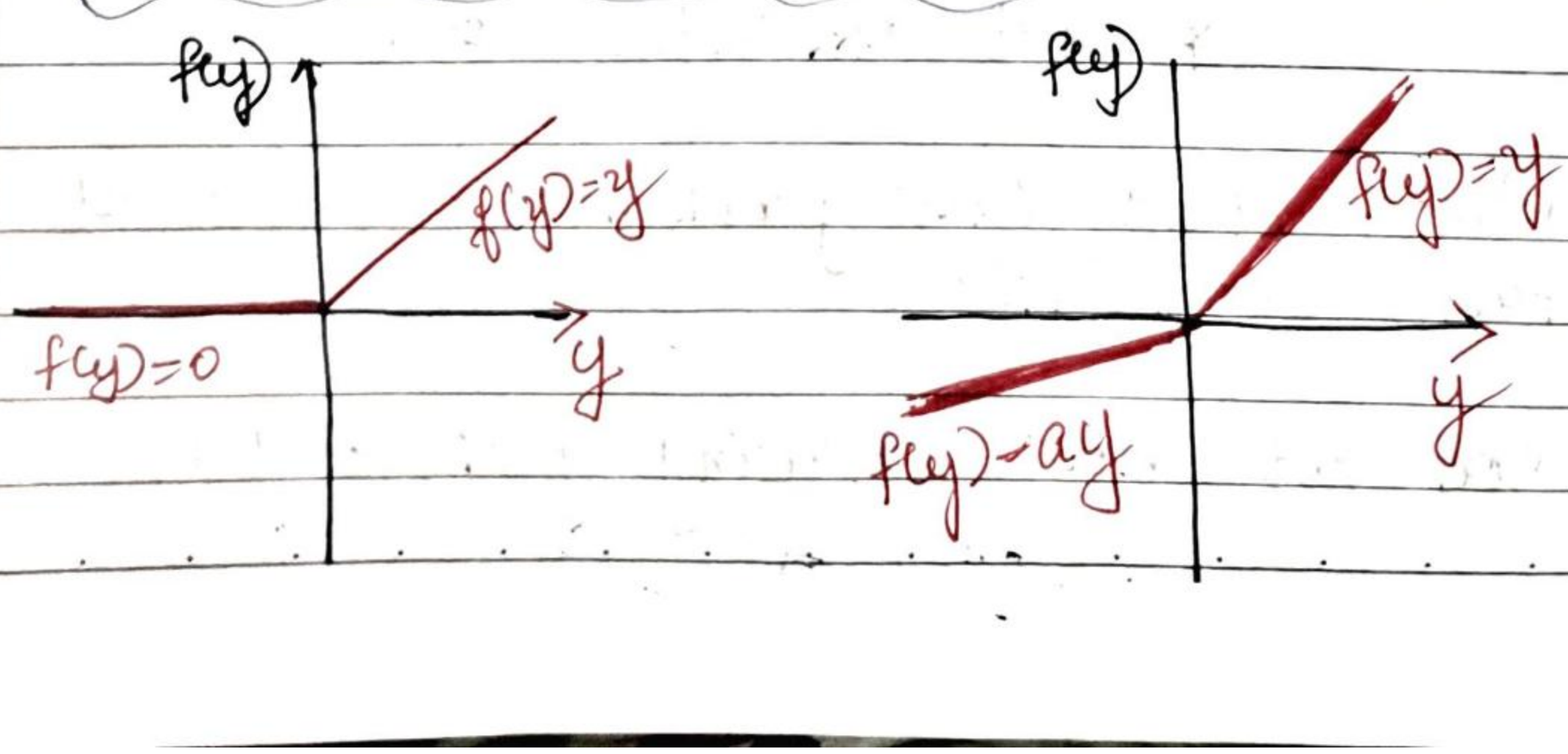
$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^x - 1), & \text{otherwise} \end{cases}$$



* ELU has all the advantages of ReLU
* No Dead ReLU issues
* The mean of the o/p is close to 0, zero-centered

* Introduces longer computation time because of the exponential operation

PReLU (Parametric ReLU)



URBAN EDGE

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

If $\alpha = 0.01$ → LeakyReLU
If $\alpha = 0$ → Relu.
α = learning Parameter

Or

$$f(y_i) = \begin{cases} y_i, & \text{if } y_i > 0 \\ \alpha_i y_i, & \text{if } y_i \leq 0 \end{cases}$$

* PRelu is an improved version of Relu.

SELU (Scaled Exponential Linear Unit)

* If we need to build a neural n/w composed exclusively of a stack of dense layers, and if all hidden layers use the SELU activation fn, then the n/w will self Normalize.

↓

* The o/p of each layer will tend to preserve mean 0 and standard deviation 1 during training, which solves vanishing/exploding gradient problem.

Conditions for self normalization

* I/p features must be standardized (mean 0, SD = 1).

* Every hidden layer's weights must also be initialized using the can normal initialization

* The n/w's architecture must be sequential

* All the layers must be dense

~~SE~~

$$f(x) = \lambda x \text{ if } x \geq 0$$

~~$f(x) = \lambda x e^{x-1}$~~

$$f(x) = \lambda \alpha (e^x - 1) \text{ if } x < 0$$

* α and λ (predetermined values)

$\alpha \approx 1.6733$ and $\lambda \approx 1.0507$.

URBAN EDGE ___/___/___

Batch Normalization:-

* One common problem when training a deep neural network is ensuring that the weights of the network remain within a reasonable range of values.

* If the weights start to become too large, network is suffering from exploding gradient problem.

  ↳ As errors are propagated backward through the n/w, the calculation of the gradient in the earlier layers can sometimes grow exponentially large, causing wild fluctuations in the weight values.

  ↳ If loss function starts to return NaN (not a number), weights have grown large enough to cause an overflow error. It won't appears as we start training the n/w. But as the n/w trains and the weights move further away from their random initial values, our assumptions may start to break down. This phenomenon is known as Covariance shift.

* Batch normalization is a solution for this covariance shift problem.

↳ A batch normalization layer calculates the mean and standard deviation of each of its input channels across the batch and normalizes by subtracting the mean and dividing by the standard deviation.

↳ There are two learned parameters for each channel
- Scale [gamma]
- Shift (beta)

↳ The o/p is a normalized input, scaled by gamma and shifted by beta

Normalization of the input:-

Normalization is the process of transforming the data to have a mean zero and standard deviation one.

① $\mu = \frac{1}{m} \sum h_i$; m = no: of neurons at layer 'h'.

[Batch input from layer 'h' calculate the mean of hidden activation]

② Calculate the standard deviation of the hidden activations

$$\sigma = \left[\frac{1}{m}\sum(h_i - \mu)^2\right]^{1/2}$$

③ Normalize the hidden activations using the above values.

↳ Substract the mean from each input and divide the whole value with the sum of standard deviation and the smoothing term (ε).

↓

used for stopping a division by a zero value

$$h_i(norm) = \frac{(h_i - \mu)}{\sigma + \varepsilon}.$$

④ Final operation :- Scaling and shifting of the i/p

↳ 2 Components → γ (gamma) ← scaling
→ β (beta) ← shifting

$$h_i = \gamma h_i(norm) + \beta.$$

Advantages of Batch Normalization :-

↳ Speed up the training

↳ Handles internal Covariance shift

(dog example)

Dropout Layers :-

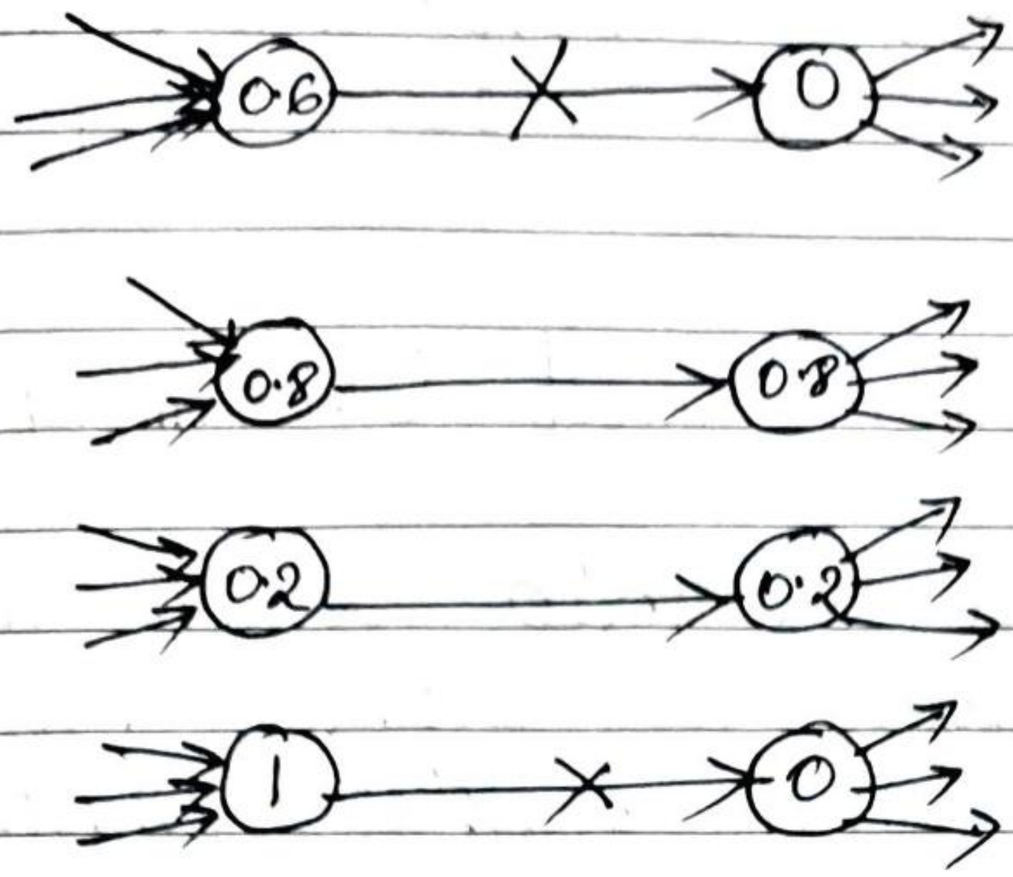
* A succesful machine learning algorithm ensures that it generalizes to unseen data, rather than simply remembering the training dataset.

* If an algorithm performs well on the training dataset, but not on the test data, then says that the model is suffering from overfitting.
To overcome this, we use regularization technique, which ~~ensure~~ ensures that the model is penalized if it starts to overfit.

* For deep learning, regularization is achieved through dropout layers.

* During training, each dropout layer choose a random set of units from the preceding layer and

sets their output to zero



Dense layers

Dropout layers

* This approach ensures that the n/w doesn't become overdependent on certain units or group of units, therefore knowledge is more evenly spread across the whole n/w. This makes the model much better at generalizing to unseen data, bcz the n/w has been trained to produce accurate predictions even under unfamiliar conditions, such as those caused by dropping random units.

* Dropout layers are used most commonly after Dense layers since these are most prone to overfitting due to the highest no: of weights.

* Batch Normalization also has been shown to reduce overfitting, and therefore many modern deep learning architectures don't use dropout at all.

Underfitting and Overfitting :-

* Underfitting is a scenario where a data model is unable to capture the relationship b/w the input and output variables accurately, generating a high error rate on both training set and unseen data.

* Underfitting occurs when a model is too simple, which can be result of a model needing more training time, more input features or less regularization

Overfitting

when the n/w tries to learn too much or too many details in the training data along with the noise from the training data which results in poor performance on unseen or test dataset when this happens the n/w fails to generalize the features / patterns

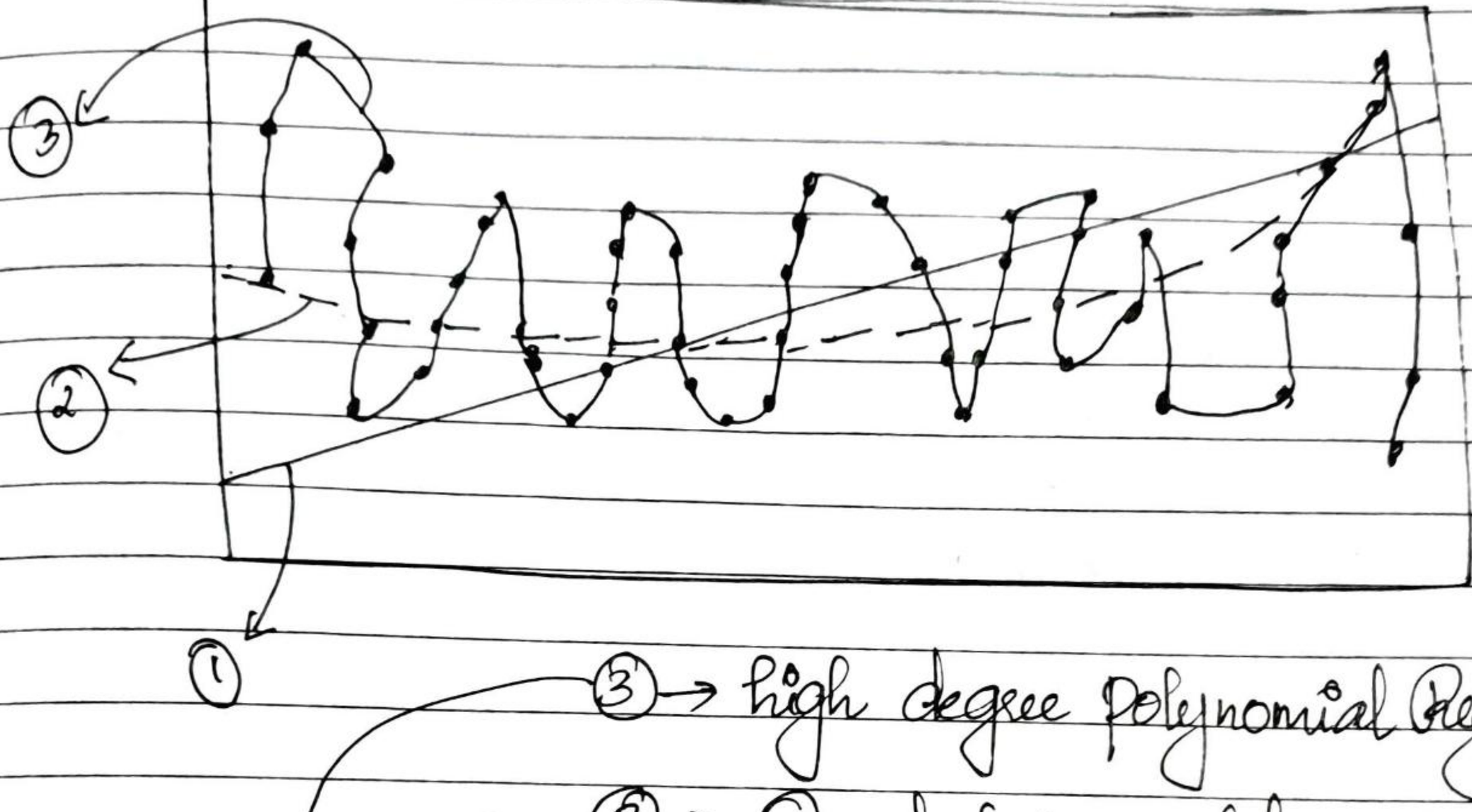
URBAN EDGE

found in the training data.

* Overfitting during training can be spotted when the error on training data decreases to a very small value but the error on the new data or test data increases to a large value.

Training and Validation Curves:

Eg for overfitting and underfitting problem:-



③ → high degree polynomial Regression
② → Quadratic model → generalized form
① → linear model

Overfitting

suffers from underfitting

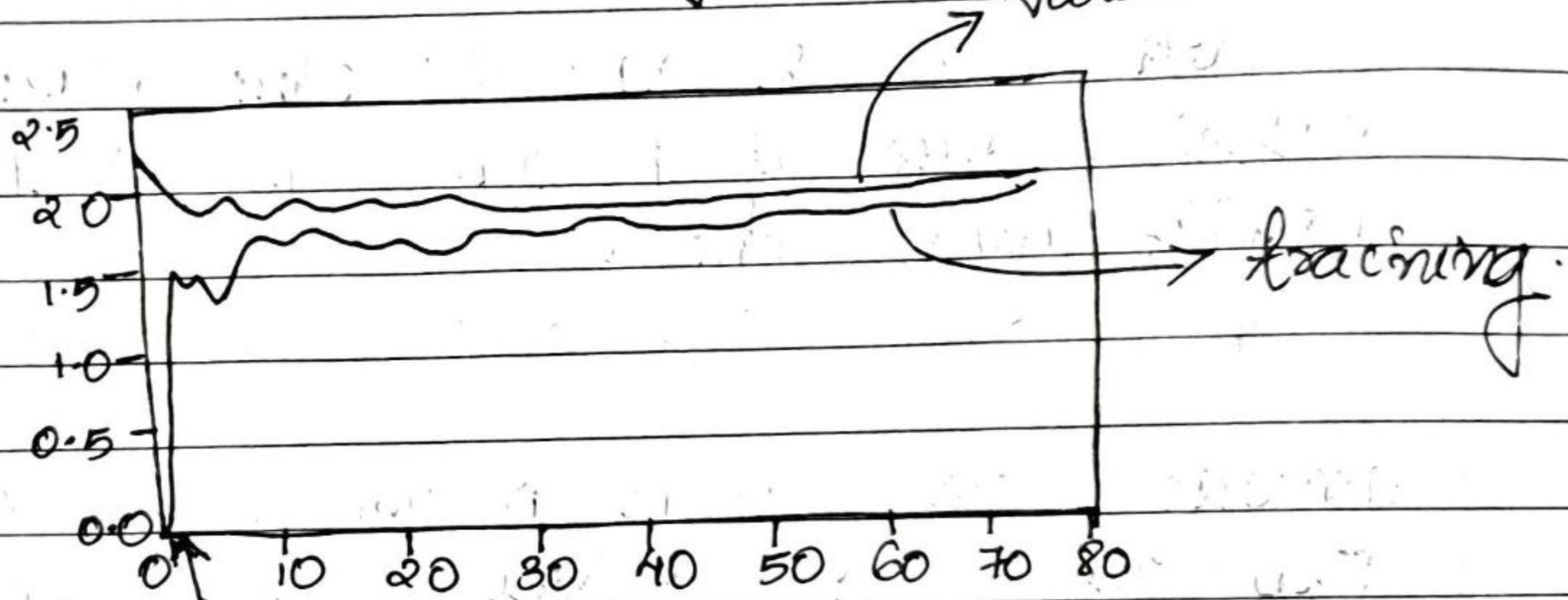
- To find out model's generalization performance we can make use of cross-validation.
  and another important method is to look at the learning curves.

Learning Curves:-

* Learning Curves are plots of the model's performance on the training set and the validation set as a function of training set size.

* To generate the plots, simply train the model several times on different sized subsets of the training set.

Let us consider an eg:-



* When there are just one or two instances in the training set the model can fit them perfectly, which is why the curve starts from zero.

* When new instances are added to the training set, it becomes impossible for the model to fit the

tracining data perfectly: because the data is noisy and bcz it is not linear at all.
∴ So the error on the tracining data goes cep centil it reaches a plateau, at which point adding new instances to the tracining set doesn't make the average error much better or worse

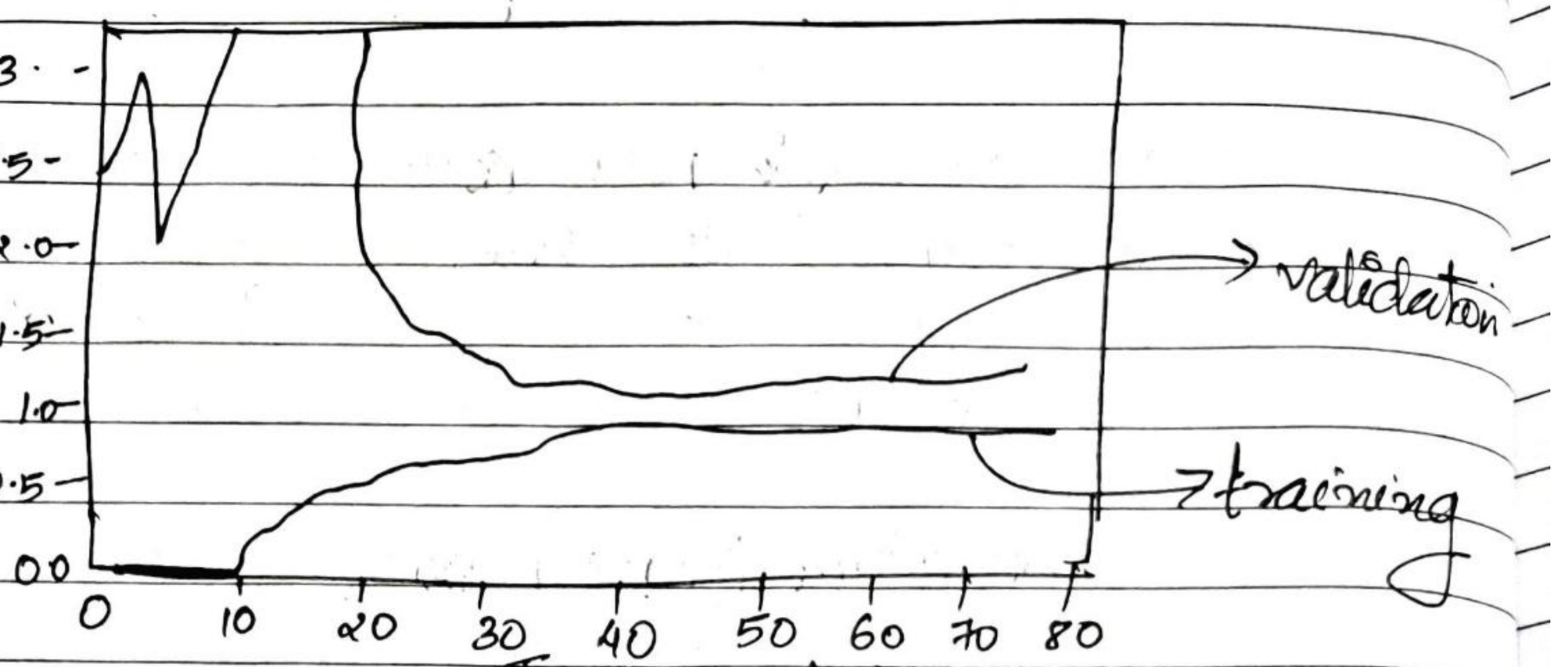
On Validation:-

When the model is trained on very few tracining instances, it is incapable of generalizing properly, which is why the validation error is initially quite big.

When with more tracining example, model learns and thus the validation error slowly goes down.

In this example, error ends cep at a plateau, very close to the other curve
This kind of model, from the learning curve we can understood its an underfitting model.

* If your model is underfitting the tracining data, adding more tracining examples will not help. We need to use a more complex model or come up with better features *



* The error on the training data is much lower than

* Gap b/w the curves. This means that the model performs significantly better on the training data than on the validation data, called Overfitting

* When training set size gets increasing, the two curves would continue to get closer.

To improve an overfitting model is to feed it more training data until the validation error reaches the training error *

URBAN EDGE ___/___/___

Bias / variance trade-off.

2 relavant topics: overfitting and underfitting.

Models generalization error can be expressed as the sum of three very different errors.

① Bias :-

* Generalization error is due to wrong assumptions
  eg:- assuming that the data is linear when it is actually quadratic.
* A high bias model is most likely to underfit the training data

② Variance :-

* Due to model's excessive sensitivity to small variations in the training data.

* A model with many degrees of freedom, likely to have high variance;
  ↳ Overfit the training data

③ Irreducible Error :-

* Due to noisiness of the data itself.
* Only way to reduce this part of the error is to ~~clean~~ clean up the data: (eg:- detect and remove outliers)

URBAN EDGE

Tradeoff

Model Complexity ↑ (increases)
↳ Variance (↑)
Bias (↓)

Models' Complexity (↓)
↳ Variance (↓)
Bias (↑)

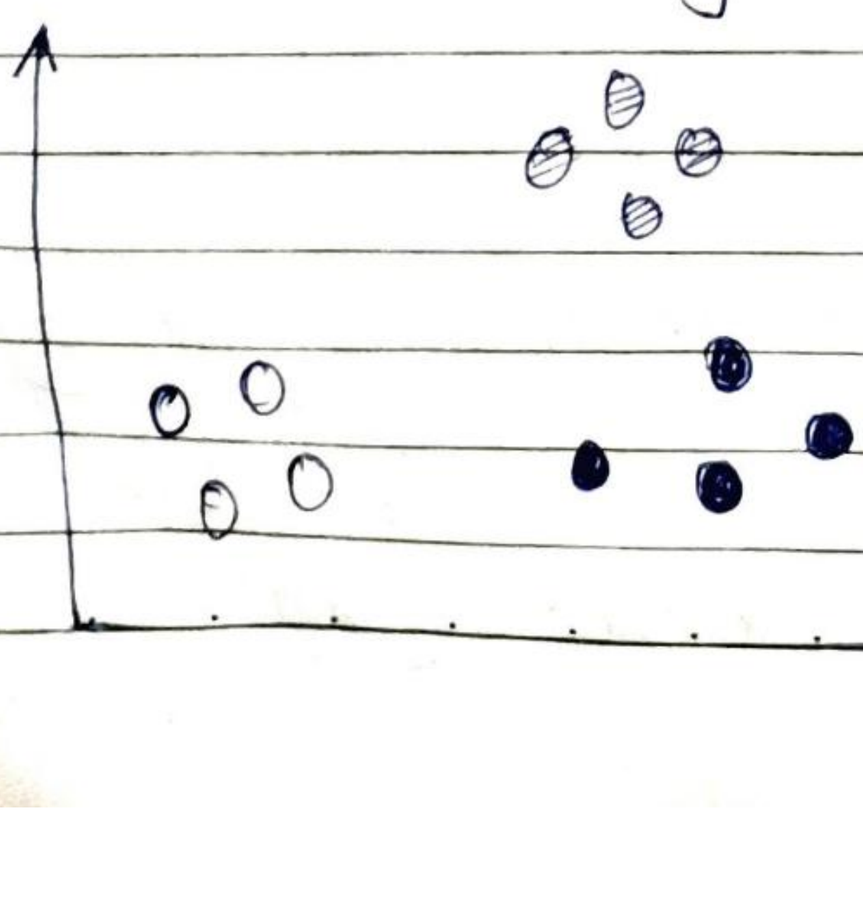
T-SNE - [t-distributed Stochastic Neighbour Embedding]

* Method for dimensionality reduction, used mainly for visualization of data in 2D or 3D maps.

(PCA - Linear Connections)

* This method can find non-linear connections in the data and therefore it is highly popular.

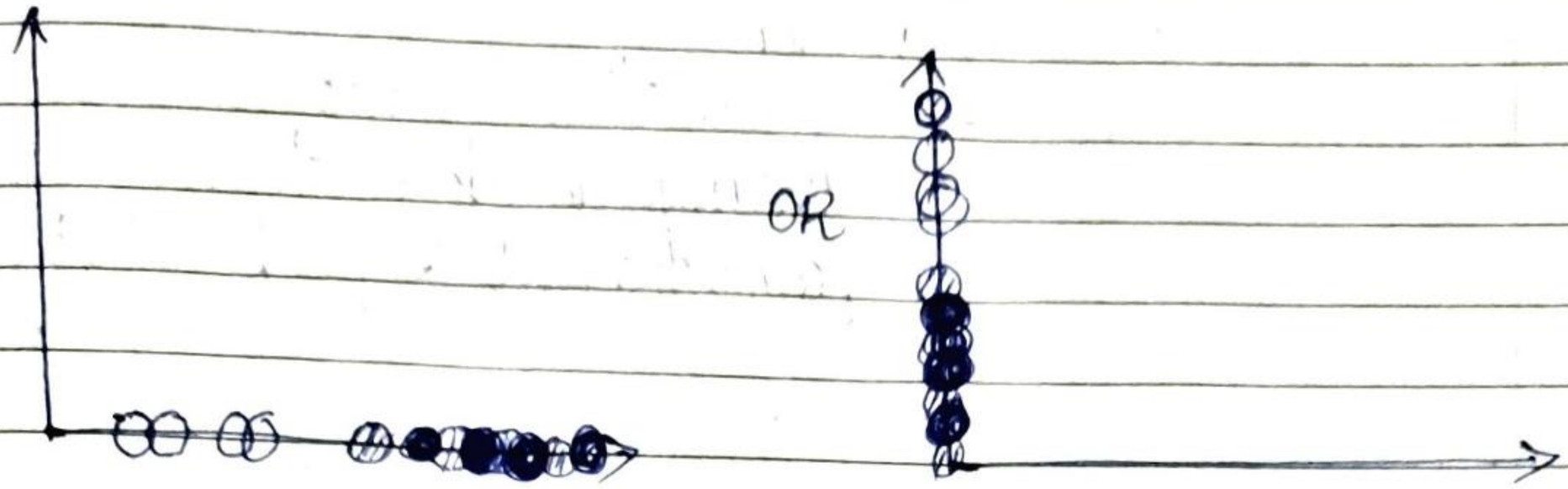
* Consider an example (2D)



→ Each pattern represents a cluster
→ Each cluster has a different density

URBAN EDGE

* Simply project the data onto (i below figure) one of its dimensions, we can see an overlap of atleast two of the clusters



= T-SNE algorithm deals with this problem !!!!

Using the below steps:-

① Calculating a joint probability distribution to represent the similarity b/w the data points.

② Creating a dataset of points in the target dimension and again calculate the joint probability of that distribution

③ Both joint probabilities should be similar in high dimensional data and in low dimensional data

Applicable for original dataset

Algorithm:-

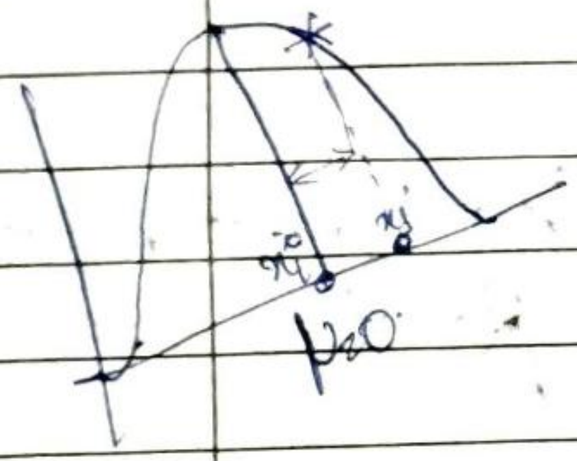
First Stage * Calculating the Euclidean distances of each point from all the other points.

- Take these distance and transforming them into conditional probability

↓

Calculating how similar every points in the data ie how likely they are to be neighbours.

Gaussian



* Conditional probability of point 'xj' given 'xi' is represented by a Gaussian centered at 'xi' with a standard deviation of 'σi'

* ↳ Mathematically,

Gaussian

$$P_{j|i} = \frac{\exp\left(-\|x_i - x_j\|^2 / 2\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-\|x_i - x_k\|^2 / 2\sigma_i^2\right)}$$

dividing by the sum of all the other points placed at the gaussian centered 'xi' is that we may deal with clusters of different densities

and case about the density of each clusters but to visualize them as clusters

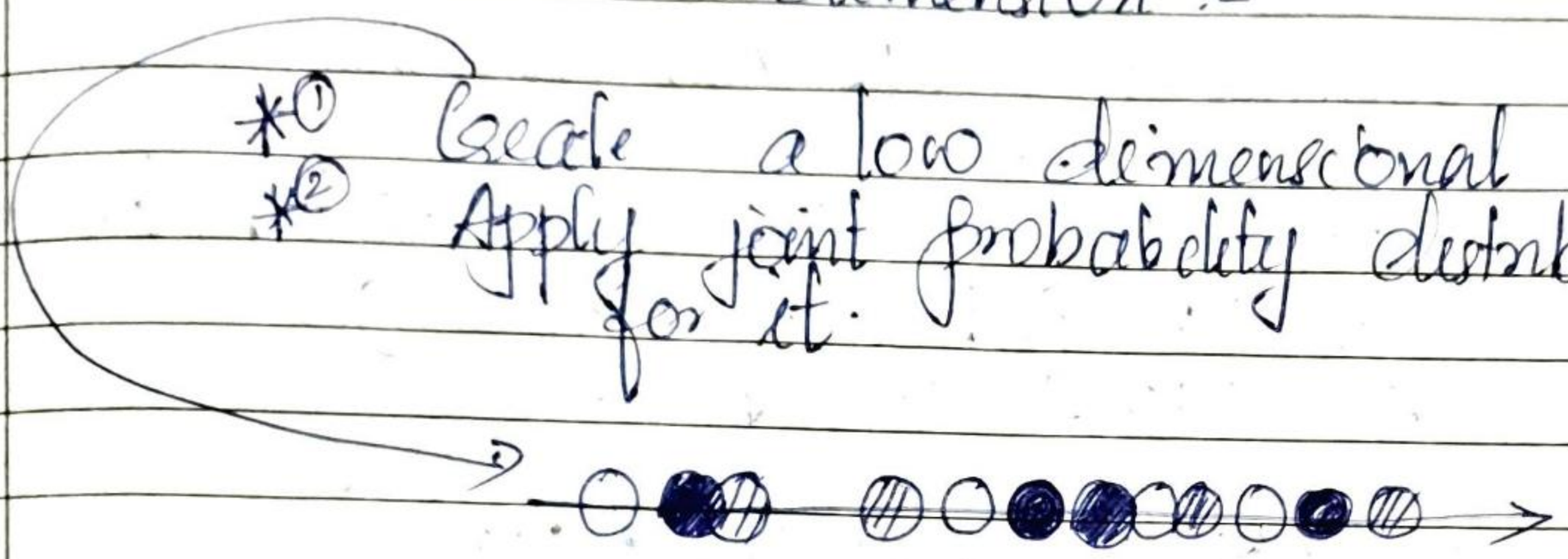
* From the conditional distributions, we need to calculate the joint probability distribution

$$P_{ij} = \frac{P_{j|i} + P_{i|j}}{2n}$$

$2n \rightarrow$ no: of datapoints

Second stage :- Creating data in a low dimension :-

*① Create a low dimensional space
*② Apply joint probability distribution for it.



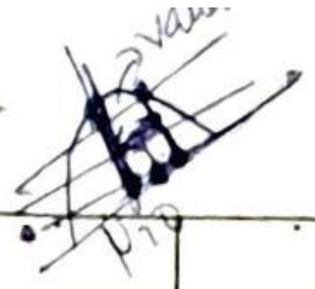
Applying | calculating joint probability distribution using t-distribution instead of gaussian distribution

probability

$$q_{ij} = \frac{1 + \left(\|y_i - y_j\|^2\right)^{-1}}{\sum_{k \neq l} \left(1 + \|y_k - y_l\|^2\right)^{-1}}$$

Advantage of t-distribution :

① → Moderate distance b/w points in the high dimensional space to become extreme in the low dimensional space. prevents "crowding" of the points in the lower dimension.



Ques 4.

Third stage:-

Apply Kullback-Leibler divergence (KL divergence) → is a measure of how much two distributions (such as joint probability distribution of the data points in the low dimension and the original dataset) are different from one another.

→ Let P and Q are distribution on the probability space X, then 'KL' divergence

$$D_{KL}(P||Q) = \sum_{x \in X} P(x) \log\left(\frac{P(x)}{Q(x)}\right)$$

↳ when the distributions are similar
↳ value of KL divergence is smaller, reaching zero when the distributions are identical

Overall

** Changing lower dimension to similar to original dataset is done through gradient descent.

** Cost fn is calculating through KL divergence.

URBAN EDGE ____/____/____

$$C = KL(P||Q) - \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

Parameters in the model

~~weight~~, ~~bias~~ * parameters for gradient descent like learning rate.

* Perplexity (for t-SNE)

used for choosing the std deviation (σ_i) of the Gaussian