Deep learning

UNIT - I

1) Biological neuron
2) artificial neuron
3) Difference b/w them.
4) Different hyper Parameters
5) Limits of Traditional Computer Program.
6) feed-foward neural network
7) Different Activation functions : Sigmoid, Tanh, Relu ...
8) Perceptron → working. → and / or logic Example.
9) Lineas Perceptron as Neurons. & limitations
10) Back Propagation algorithm, in Reducing Error.
11) Gradient descent with Sigmoid Neuron
    ↳ Derivation
12) delta Rule for training linear neurons.
13) Preventing Overfitting, in DL methods & Demonstration.
14) Stochastic & minibatch Gradient.

↓

Comparision

15) Mechanics of ML.

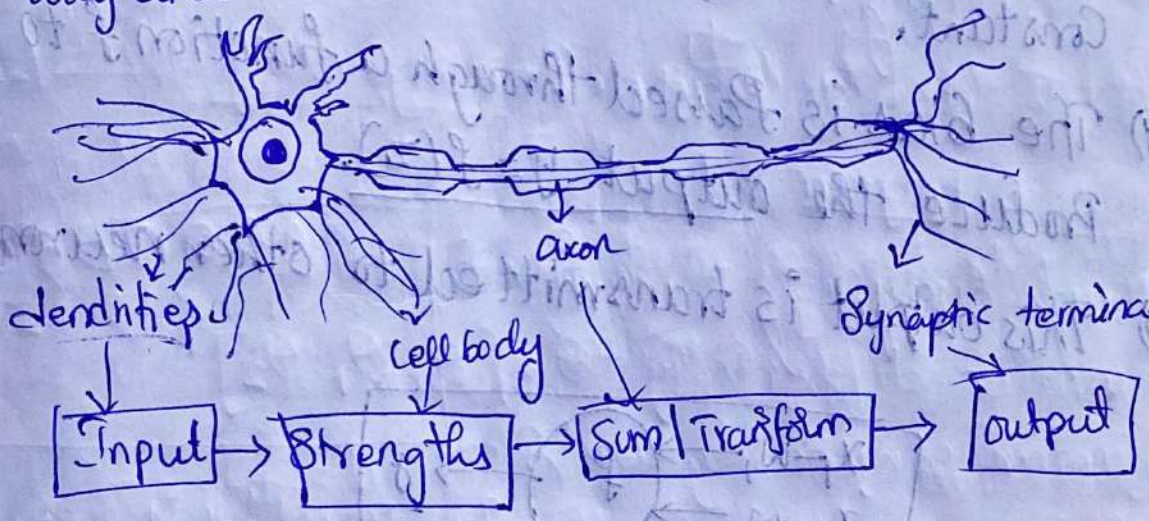
Unit 1 Chapter 1 (Q1, 2, 4, 5, 6, 7, 8, 9, 15)

1) Biological Neurons:

1) The foundational unit of human brain is the neuron.

*) It is the massive biological network that enables us to experience outside world.

*) As it's core, the neuron is optimized to receive info from other neuron, process the info in a unique way and send it's results to other cell.



→ Explain Process:-

1) The neuron Recieves input along dendrites.

2) Each of incoming connection is dynamically Strengthened & Weakened based on how it is used.

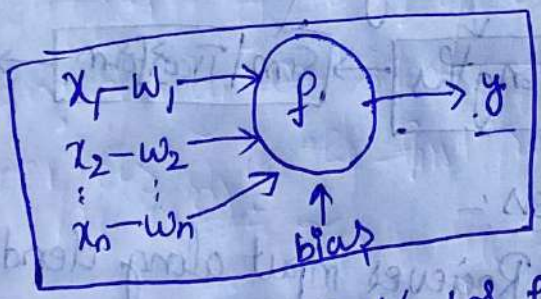
3) After being weighted by the strength of their respective connection, the input goes to cell body.

4) This Sum is then transformed into a new signal that's propagated along the cell axon's & send off to other neuron,

(2) Artificial neuron:

4708A

1) Such as Biological neuron the (A.N) takes in some no of inputs $x_1, x_2, \ldots x_n$ each of which is multiplied by weight, $w_1, w_2, w_3 \ldots w_n$.
2) These weighted inputs are, as before, summed together to produce the logit of the neuron, $z = \sum_{i=0}^{n} w_i x_i$.
3) The logit is also called as bias which is constant.
4) The bias is passed through a function f to produce the output $y = f(z)$.
5) This output is transmitted to other neurons.



We can express this in vector form

input vector $x = [x_1, x_2, \ldots x_n]$

weight vector $W = [w_1, w_2 \ldots w_n]$

Expression: $y = f(x \cdot W + b)$ where b is bias term

3)

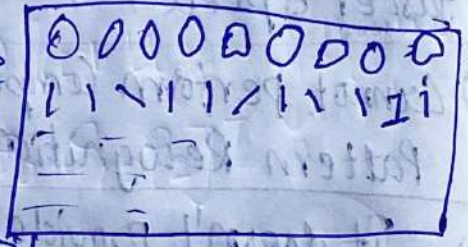
Artificial Neuron	Biological Neuron
1) It is a mathematical model which is mainly inspired by the biological neuron system in the human brain.	It is also composed of several Processing Pieces known as Neurons.
2) Its Processing was Sequential and centralized.	It Process the info in a Parallel and distributive manner.
3) Small in Size	large in size.
4) It process the info at a faster speed	It process the info at a slow speed.
5) Cannot perform Complex Pattern Recognition	Perform Complicated tasks.
6) It doesn't provide any feedback	It Provides feedback.
7) No fault tolerance	fault tolerance is their.
8) Operating Environment is well defined and well-constrained	Operating Environment is poorly defined and Uncon-strained.
9) Reliability: It is very Vulnerable	It is robust.
10) Response time is measured in milliseconds.	It's Response time is measured in nanoseconds
11) It has very accurate Stru-cture	They are tolerant & ambiguity.

4)

5) Limits of Traditional Computer Programs.

* Traditional computer programs are designed to be very good at two things
  1) Performing arithmetic really fast.
  2) Explicitly following a list of instructions.
* The T.C.P are good at heavy financial number crunching.

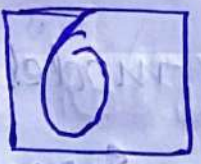
Let us take an Example: Recognizing hand written digits Program



although Every digit in the figure is written differently the human can Easily identify the number as 0 in the first row, 1 is in second row etc.,

But if we are said to write Program to automatically read Someone's handwriting, it will be very difficult.

If we might State that we have a "zero" if our image only has a Single, closed loop. but this isn't really a Sufficient condition what if Someone write a messy zero, it will be a very difficult task to identify it.

Like the fig describes 

It can't defo identify it as 0 or 6 in this case if we even keep the distance between them in the loop.

(15) Mechanics of Machine Learning

→ To tackle the above Handwritten problem we have to use different kind of approch..

→ from childhood to adulthood we often learn things and we learn what is wrong and we will learn from it and train from it.

→ we learn how to multiply numbers, solve Equations, take derivates by set of instructions.

→ we learned to recognize a dog by being shown multiple examples and being corrected when we made the wrong guess. if our Parents say we are wrong we will modify our model to incoperate this new information.

→ Deep Learning is a subset of a more general field of AI called ML. which is predicted on this idea of learning.

→ we give model to train and learn instead of set of large instructions.

Lets define our model to be a function $h(x, \theta)$.

The Input (x) is an Example Expressed in Vector form.

$\theta \rightarrow$ Vector of the parameter that model use.

Let take an Example: ~~Sleep~~ Predict Exam Performace on ~~Sleep~~ no of hours of Sleep.

We collect a lot of data for each data Point $x = [x_1 \; x_2]^T$, $x_1 \rightarrow$ no of hours of Sleep.

$x_2 \rightarrow$ no of hours we ~~spent~~ Studying.

Our goal, the model $h(x, \theta)$ with Parameters Vector $\theta = [\theta_0 \; \theta_1 \; \theta_2]^T$

Such that:

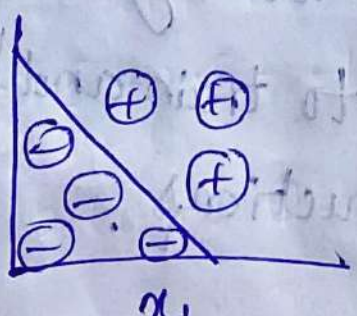
$$h(x, \theta) = \begin{cases} -1 \text{ if } x^T \cdot \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} + \theta_0 < 0 \\ 1 \text{ if } x^T \cdot \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} + \theta_0 \geq 0 \end{cases}$$

-1: bellow average

1: Otherwise

This model is called linear Perceptron

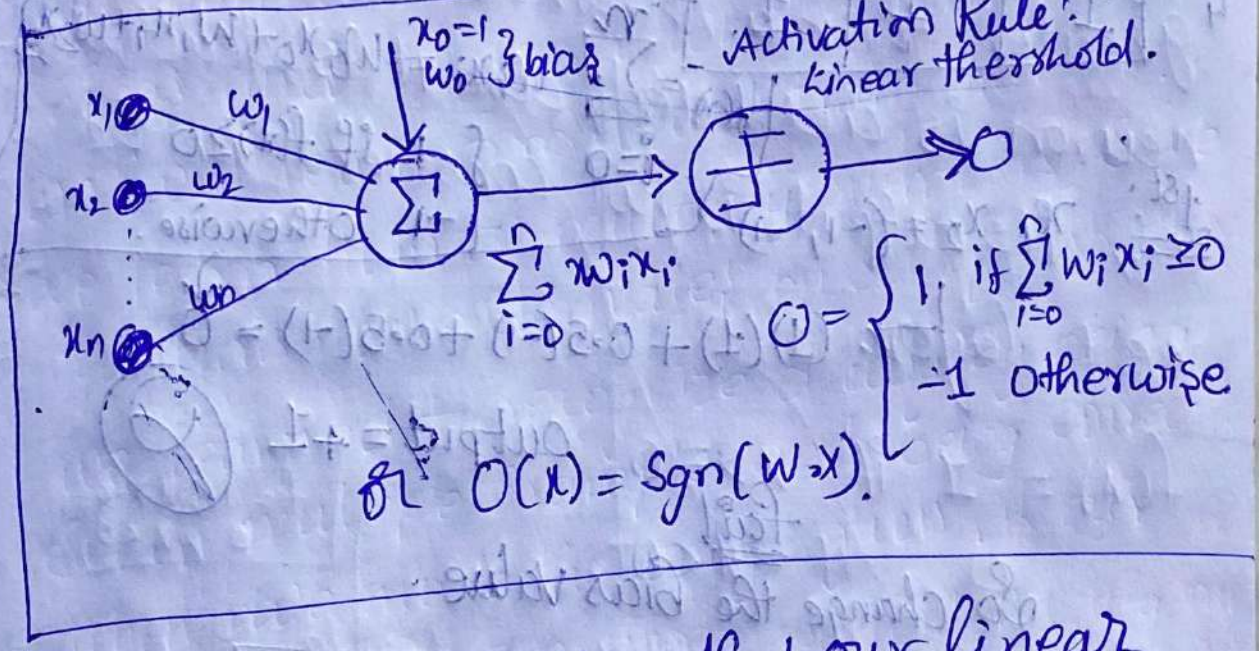
Sample data:



⑦ Linear perceptron as Neurons & limitations:

A linear perceptron or perceptron is a ML algorithm for Supervised learning of various binary classification tasks.

The Single layer neural network has four Parameters: input values, weights, Bias netsum and an activation function.



It is very easy to show that our linear Perceptron & the neuronal model are Perfectly Equivalent. And it is quite Simple to show that Singular neurons are Strictly more Expressive as linear perceptron.

Trying the Linear Perceptron by using and gate:

	x_1	x_2	y
1	-1	-1	-1
2	1	-1	-1
3	-1	1	-1
4	1	1	1

Let us take

$W_0 = 1$ (bias) $x_0 = 1$

$w_1, w_2 = 0.5$ (assum)

We know formula!

$$f(x) = \sum_{i=0}^{n} w_i x_i = w_0 x_0 + w_1 x_1 + w_2 x_2$$

$$\begin{cases} 1 & \text{if } f(x) \geq 0 \\ -1 & \text{otherwise.} \end{cases}$$

1st: $x_1, x_2 = (-1, -1)$

$\hookrightarrow (1)(1) + 0.5(-1) + 0.5(-1) = 0$

output = +1 ✗

fail

So change the bias value.

$W_0 = -1$ | $W_1, W_2 = 0.5$ | $x_0 = 1$

$P(-1,-1) \rightarrow -1 - 0.5 - 0.5 = -2 < 0$ output = -1 ✓

$P(1,-1) \rightarrow -1 + 0.5 - 0.5 = -1 < 0$ output = -1 ✓

$P(-1,1) \rightarrow -1 - 0.5 + 0.5 = -1 < 0$ output = -1 ✓

$P(1,1) \rightarrow -1 + 0.5 + 0.5 = 0 = 0$ output = 1 ✓

Limitations:-

1) Lack of Complexity: They are incapable of handling complex patterns

2) Limited Expressiveness:

3) No feature Hierarchy: due to linear nature they cannot form such hierarchies.
(Simple to Complex)

4) Inadequate for Complex tasks:

5) Overfitting to linear Assumptions:

Activation functions:

The Activation function is one that outputs a smaller value for tiny inputs and a higher value if it's inputs are greater than threshold.

There are two type of activation functions:

1) linear Activation function.

2) Non-linear Activation function.

Linear Activation function:

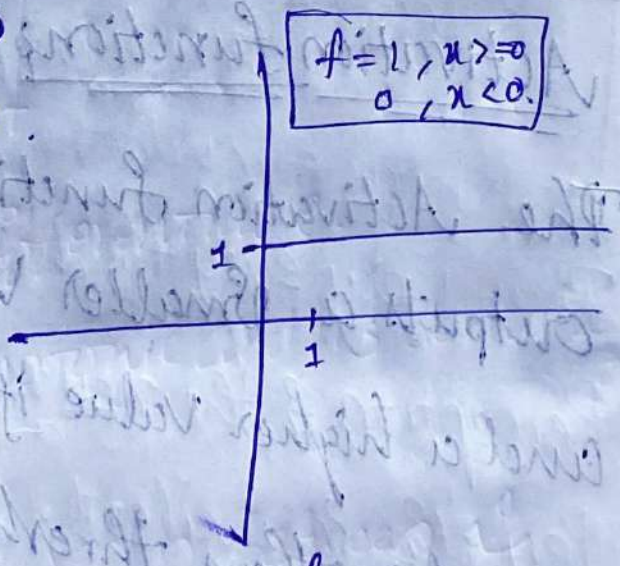
The function is linear.

*) Binary Step Function:

If the input to the activation function is greater than a threshold then the neuron is activated, else it is deactivated. ie., its output is not considerd for the next layer.

$f(x) = 1$ if $x >= 0$

0 if $x < 0$.



drawbacks:

*) It can be only used while creating a binary classifier

*) This function is not useful when there are multiple classes in the target variable.

**)

Non Linear Activation functions:

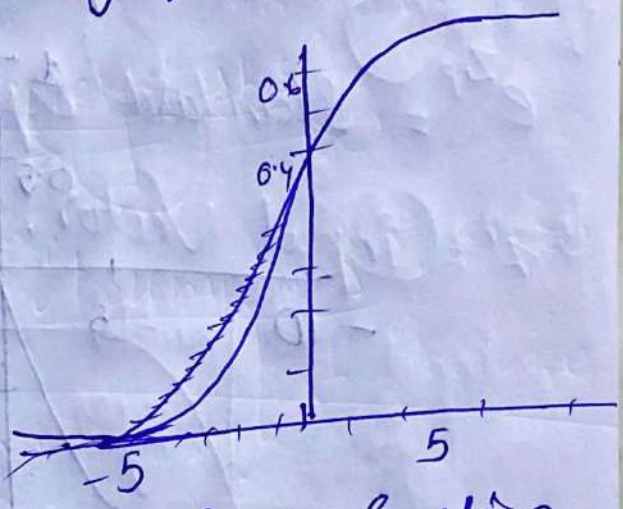
~~The input of~~ The functions will be non linear

1) Sigmoid :-

A Sigmoid function is graphed in "S" shape.

It uses the function:

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



*) when the logit is very small, the output of a logistic function is very close to "0".

*) when the logit is very large the output of the logistic function is close to "1".

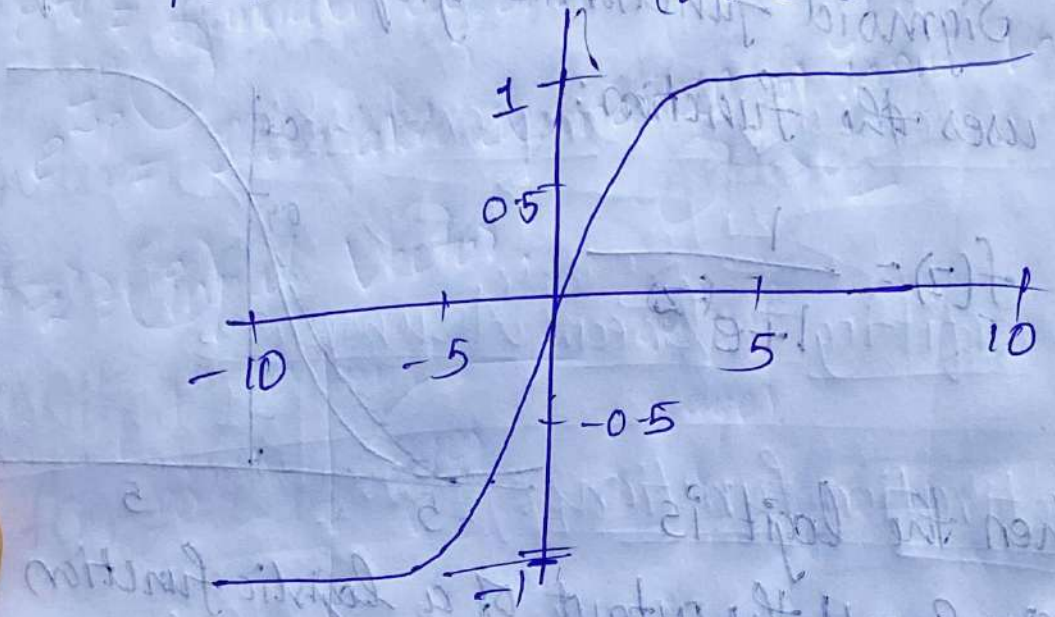
*) In b/w them, the function assumes an S-shape.

(2) Tanh function:

The Tanh function use a similar kind of S-shaped non-linearity, but insted ranging from 0 to 1, the output of tanh function ranges from -1 to 1.

It uses:

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



→ The function is differentiable.

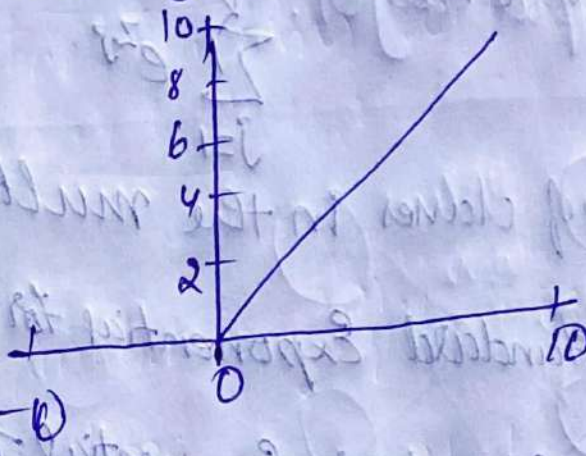
→ The function is monotonic and its derivative not.

→ It is mainly used in between of classes.

3) RELU activation function

It is Rectified Linear unit

It uses the function $f(z) = \max(0, z)$, resulting a characteristic hockey-stick-shaped response.



It Ranges from 0 to ∞.

The function is monotonic.

4) Softmax Function:

→ The Softmax function can be used for multiclass classification problems.

→ It is described as combination of multiple Sigmoids.

→ Sigmoid returns values b/w 0 and 1, which can be treated as probabilities of a data Point belonging to a particular class.

This function returns the probability for a data point belonging to each individual class.

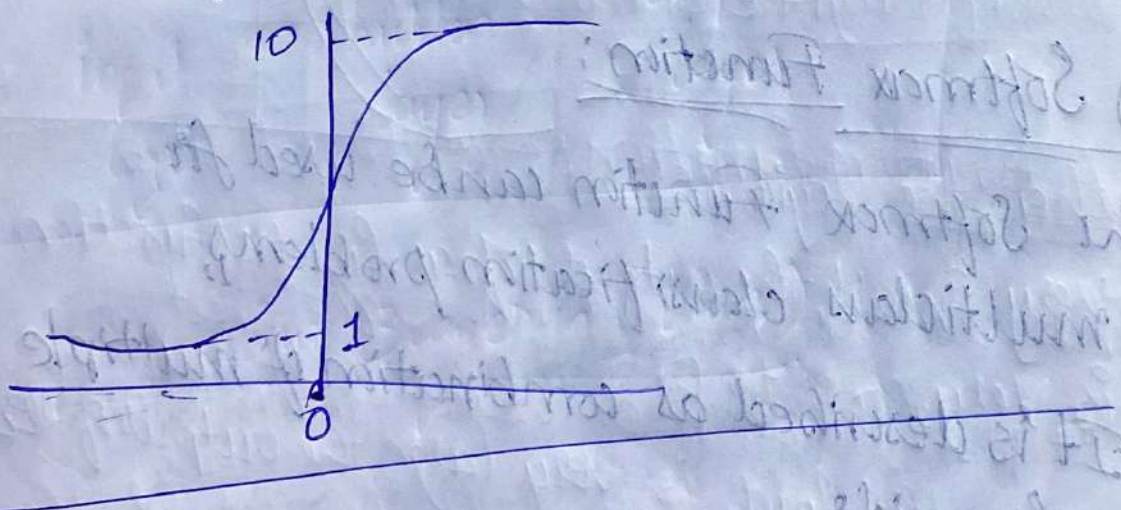
$$\sigma(\vec{z}) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

K = no of classes in the multiclass classifier

e^{z_j} = Standard Exponential for output vector

e^{z_p} = Standard Exponential for input vector.

$\vec{z}$ = input Vector.



Hyper parameters: (not important)

1) The k in KNN
2) Learning rate: for training a neural Netwo
3) Train-test split ratio
4) Batch size
5) Number of Epochs
6) Braches in Decision Tree
7) Number of clusters in clustering algorithms

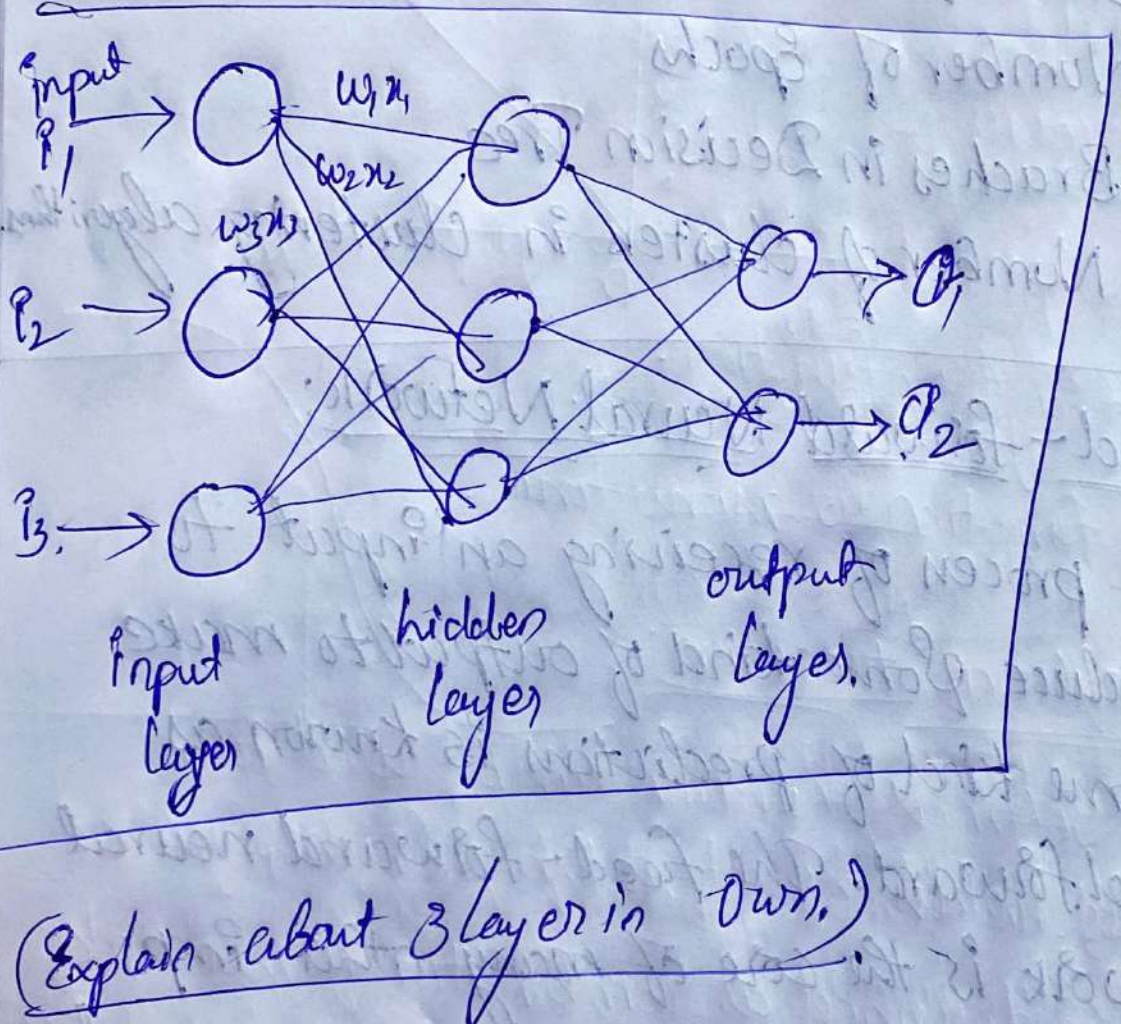
Feed-forward Neural Network:

The proces of receiving an input to produce some kind of output to make some kind of predictions is known as Feed forward. The feed-forward neural network is the core of many other imp. neural network such as Convolution neural network.

In the feed-forward neural network, there are not only any feedback loops of connections in the network. Here is an Here

there are 3 layers:

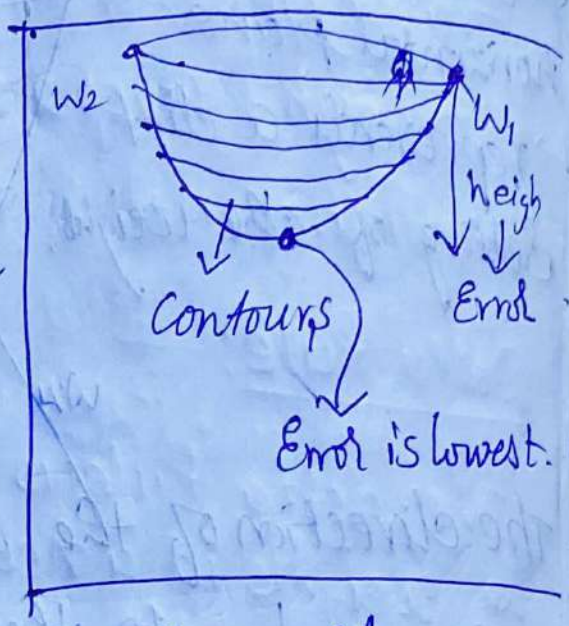
1) Input layer
2) Hidden layer
3) Output layer.



(Explain about 3 layer in own.)

Unit 1 Chapter 2

Gradient Descent:



G.D. is known as one of the most commonly used optimization algorithm to train ML models by means of minimizing errors between actual & Expected results.

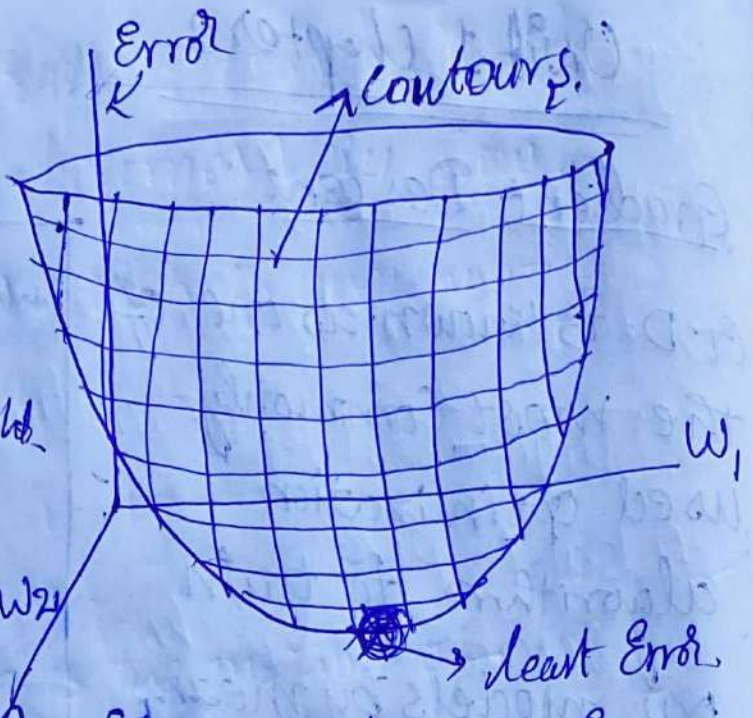
→ It helps to find the local minimum of a function

→ The main objective of using a gradient descent algorithm is to minimize the cost function using iteration.

→ The 3D space where the horizontal dimensions corresponds to the weights w_1 and w_2, and the vertical dimension corresponds to the value of the Error function E.

→ As shown in figure the circles are Elliptical "Contours". Where the minimum Error is at the center of the ellipse.

* Each point on the horizontal plane represents a diff setting of the weight.



The direction of the steepest descent is always Ire to these contours.

→ We start by initializing the weights (w_1 & w_2) randomly.

→ Evaluate the gradient at this starting position to find the direction of steepest descent.

→ Take a step in the direction of this gradient

→ After the step, you will be at the new position with a potentially lower Error.

→ Reevaluate the gradient at this new position and again move in the direction of steepest descent.

→ This Iterative Evaluating the gradient and taking steps in its directions, we

→ Progressively move closer to the point of minimum Error.

→ This Iteration process is the main in gradient descent.

The Descent Rule & Learning Rule :-

→ In neural networks, apart from weights, there are other parameters known as hyperparameters that help guide the training process. one such hyperparameter is "Learning Rate."

→ The "Learning Rate" controls how much we adjust the weights of our Networks wrt to the gradient.

→ When using the gradient descent the learning rate determines how big a steap we take at each iteration.

→ The Steepness of the Error Surface can indicate how close we are to minimum Error.

- → when Surface is flat → we are closer to minimum. Should take smaller steps.
- → Surface is steep → take larger steps.

→ If the learning Rate is too small, the training process becomes very slow.

→ if too high, there's a risk of overshooting the minimum.

Delta Rule:

The delta Rule is the method of updating the weights in a neural network.

It involves calculating the gradient of the Error function wrt. each weight.

By apping the gradient and the learning rate, the delta rule adjusts each weight to reduce the Error.

we know that:

$$\Delta W_k = -\epsilon \frac{\partial E}{\partial W_k} \quad \text{—①}$$

We also know that

$$E = \frac{1}{2} \sum (t_k - o_d)^2 \rightarrow ②$$

(t_k: target, o_d: predicted)

② in ①

$$\Delta w_k = -\epsilon \frac{\partial}{\partial w_k}\left[\frac{1}{2}\sum (t_i - o_d)^2\right]$$

$$= -\epsilon \frac{1}{2} 2(t_i - o_d) \frac{\partial}{\partial w}(t_i - o_d)$$

$$= -\epsilon (t_i - o_d) \frac{\partial}{\partial w}\left(\frac{t_i}{0} - \frac{(w_i \cdot x)}{1}\right) \quad \boxed{\because o_d = w_i \cdot x}$$

$$= -\epsilon (t_i - o_d)(-x)$$

$$\boxed{\Delta w_k = \sum_i \epsilon x (t_i - o_d)}$$

Applying this method of changing the weights at Every Iteration, we are finallyable to utilize gradient descent.

Gradient Descent with Sigmoidal Neurons

Lets recall the mechanism by which logistic neurons compute their output value from their inputs:

$$z = \sum_k w_k \cdot x_k$$

$$y = \frac{1}{1 + e^{-z}}$$

The neuron computes the weighted sum of its inputs, the bias "z". It then feeds its bias into the input function to compute y, its final output.

To do so, we start by taking the derivatives of the bias wrt input & the weight:

$$\nabla w_t = \frac{\partial E}{\partial w_k}$$

$$\therefore E = \frac{1}{2}(f(x) - y)^2$$

$$\because f(x) = \frac{1}{1 + e^{-(wx+b)}}$$

$$\frac{\partial E}{\partial w_k} = \frac{1}{2}\left[\frac{1}{1+e^{-(wx+b)}} - y\right]^2$$

$$= \frac{1}{2}\left[2x(f(x)-y) * \frac{\partial}{\partial w}(f(x)-y)\right]$$

$$\frac{\partial E}{\partial w_k} = (f(x)-y) \cdot x \frac{\partial}{\partial w}(f(x)-y) \quad \text{---(1)}$$

$$\therefore \frac{\partial}{\partial w}(f(x)-y)$$

$$= \frac{\partial}{\partial w}\left[\frac{1}{1+e^{-(wx+b)}} - y\right]$$

$$= \frac{-1}{(1+e^{-(wx+b)})^2} \frac{\partial}{\partial w}\left(e^{-(wx+b)}\right)$$

$$= \frac{-1}{(1+e^{-(wx+b)})^2} * e^{-(wx+b)} \frac{\partial}{\partial w}\left(-(\underset{1}{w}x+\underset{0}{b})\right)$$

$$= \frac{-1}{(1+e^{-(wx+b)})^2} \times e^{-(wx+b)} (-x)$$

$$= (+x) \frac{-1}{(1+e^{-(wx+b)})^2} \times e^{-(wx+b)}$$

$$(z = wx+b)$$

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

$$\frac{\partial f}{\partial w_k} = f(x) \times (1-f(x)) (-x) \rightarrow (2)$$

keep (2) in (1) $f(x)=y$

$$= [f(x)-y]\,[f(x) \times (1-f(x)) (-x)]$$

$$= -x\, f(x)(f(x)-y)(1-f(x)) \frac{\partial y}{\partial w_k}$$

$$\Delta E_k = -\epsilon \frac{\partial E}{\partial w_k}$$

$$\Delta w_k = -x\, f(x)(f(x)-y)(1-f(x))$$

$$w_{t+1} = w_t - \eta \nabla_{wt}$$

$$= w_t - \eta \sum (y - y)(t - y)$$

$$w_{t+1} = w_t - \eta (f(x) - y) f(x)(1 - f(x)) x$$

As you notice, the new modification rule is just like the delta rule, except with extra multiplicative terms included to account for the logistic component of the Sigmoidal neuron.

Backpropagation algorithm: Or reducing Error.

algorithm:

Step 1: Calculate the Error at the output

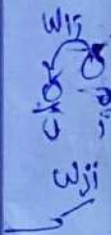
* Start with the output layer of the neural network
* for each neuron i in the output layer, calculate the Error (E) as the Sum of the Squares of the difference between the Predicted output (y_i) and the true target (t_i)

$$E = \frac{1}{2}\sum_j (t_j - y_j)^2$$

* compute the derivate of the Error wrt. each output neuron's activity.

$$\frac{\partial E}{\partial y_j} = \frac{1}{2} \times 2 (t_j - y_j) \frac{\partial}{\partial y_j}(t_j - y_j)$$

$$\frac{\partial E}{\partial y_j} = -(t_j - y_j)$$



Step 2: Back propagate the Error.

* Presume that the Error derivatives for layer j are known.
* To calculate the Error derivates for the Preceding layer i, accumulate info on how each neuron in layer "i" influences the neuron in layer "j".
* This can be done as follows using the Partial derivate of the bias wrt incoming Output data from the layer beneath is merely the weight of the connection w_{ij}.

$$\frac{\partial \epsilon}{\partial y_i} = \frac{\partial \epsilon}{\partial z_j} \frac{\partial z_j}{\partial y_i}$$

$$= w_{ji} \frac{\partial \epsilon}{\partial z_j} \rightarrow (1) \quad \left(\frac{\partial z_j}{\partial y_i} = w_{ji} \right)$$

$$\frac{\partial \epsilon}{\partial z_j} = \frac{\partial \epsilon}{\partial y_i} \frac{\partial y_i}{\partial z_j}$$

$$\frac{\partial \epsilon}{\partial z_j} = \frac{\partial \epsilon}{\partial y_i} y_i(1-y_i) \rightarrow (2)$$

(1) & (2)

$$\frac{\partial \epsilon}{\partial y_i} = \sum_j w_{ji} y_j (1-y_j) \frac{\partial \epsilon}{\partial z_j}$$

Step3: update the weights.

* After calculating the partial derivatives for all units, update the weights to minimize the error, using the learning rate "η"

$$\Delta w_{ji} = -\eta \frac{\partial \epsilon}{\partial w_{ji}} \quad \text{---(3)}$$

We know

$$\frac{\partial \epsilon}{\partial w_{ji}} = \frac{\partial z_j}{\partial w_{ji}} \frac{\partial \epsilon}{\partial z_j}$$

From (2)

$$\frac{\partial \epsilon}{\partial w_{ji}} = \frac{\partial \epsilon}{\partial y_j} y_j(1-y_j) \times \frac{\partial z_j}{\partial w_{ji}}$$

$$\frac{\partial \epsilon}{\partial w_{ji}} = \frac{\partial \epsilon}{\partial y_j} y_j(1-y_j) \times y_i \qquad \left[\because \frac{\partial z_j}{\partial w_{ji}} = y_i\right]$$

~~Now keep the~~

Now for all the training examples, sum up the partial derivatives to adjust the weights:

$$\Delta w_{ji} = -\eta \frac{\partial \epsilon}{\partial y_j} y_j(1-y_j) \times y_i.$$

$$\boxed{\Delta w_{ji} = -\eta \sum_{x \in \text{dataset}} y_i\, y_j(1-y_j) \frac{\partial \epsilon}{\partial y_j}}$$

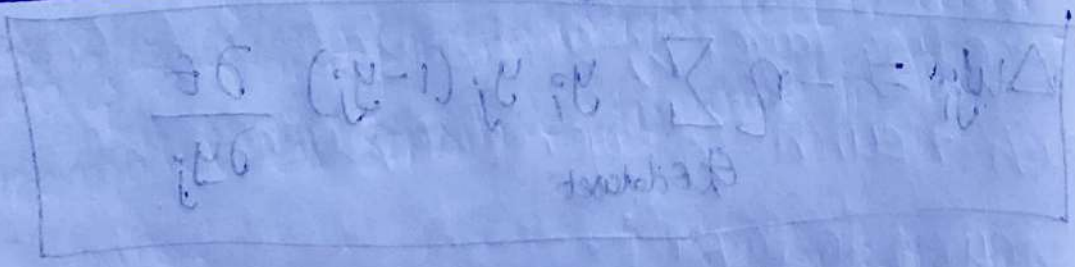
Step 4: Iterate the process:

Repeat steps 1 to 3 for each layer in the network, moving from the output layer towards the input layer.

Iterate the process across all layers for multiple epochs or until the network converges to a satisfactory level of Error.

Stochastic and Minibatch Gradient.

- after Evaluating Each individual training Example.
- It is a compromise b/w batch gradient descent & Stochastic gradient descent.
- It Computes the gradient against a Subset of the data known as Mini batch.



Different techniques to prevent overfitting.

1) ~~Regularization~~

Overfitting :-

the model becomes very good at predicting or fitting the data it has been but fails to generalize to data that hasn't been trained on.

1) **Regularization :-**

* Regularization is a technique to prevent overfitting by adding a penalty for larger weights in the objective function.

* The modified objective function becomes Error + $\lambda f(\theta)$, where $f(\theta)\uparrow$ with larger weights & λ is the regularization strength, a hyperparameter.

* Choosing λ is crucial : $\lambda = 0$ means no Regularization while a large λ might prioritize small weights over fitting the training data.

L2 Regularization (Weight Decay):

The most common regularization in ML, L2 Reg. adds $\frac{1}{2}\lambda W^2$ to the Error function for every weight "W".

It penalizes large weights, Preferring more evenly distributed weight values and Prevents Overfitting by encouraging the whole network to use all inputs rather than a few inputs a lot.

During gradient descent, L2 causes weights to decay linearly towards zero.

3) L1 Regularization:

* Adds the term |W| for each weights in the network.
* Leads to Sparsity in the weight vector, making the model less sensitive to noise in the inputs.
* useful for feature Selection as it indicates which features ~~descent~~ are most important for Predictions.

4) Max Norm Constraints:-

* Enforce an absolute upper bound on the magnitude of the weights vectors.
* If the weight exceeds the bound during the gradient descent update, it is projected back onto a predefined bound.
* Prevents weights from growing too large, ensuring stability in the learning process.

5) Droupout:-

* A technique where each neuron is kept active with a certain probability P during training, & set to zero otherwise.
* makes the network less reliant on any single neuron, effectively combining many different neural network architectures.
* Dropout can prevent Overfitting by ensuring the network remains accurate. Even when certain neurons are dropped.

6) Inverted Dropout:

* A variation of dropout where the neuron's output is scaled at training time, rather than test time.
* Ensure that a test time, the neuron's output do not need to be scaled, maintaining the expected output level.

→ The Autoencoder can effectively summarize data points when all relevant info is present in the individual data points themselves.

Chapter 1

1) Vanilla Deep Neural networks

Deep learning aims to automate feature selection, crucial for computer vision tasks.

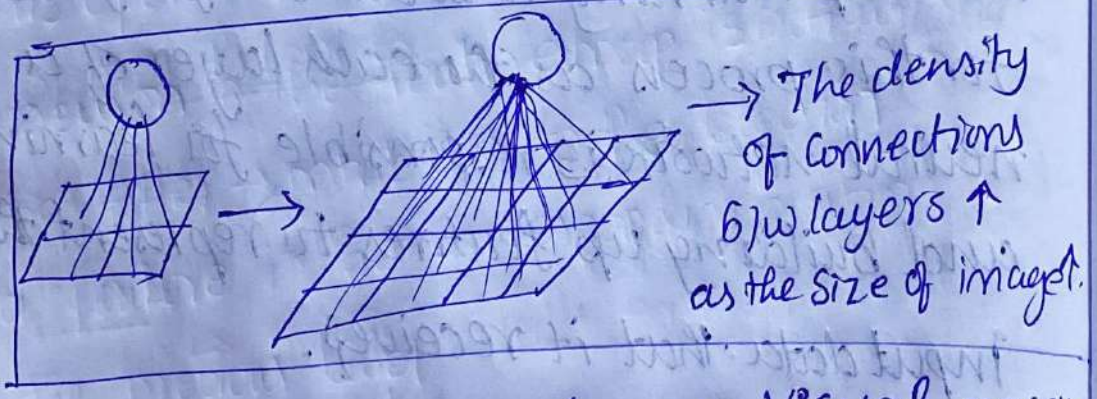
The Deep learning network are prefects for this process as the each layer of a neural network is responsible for learning and building up features to represent the input data that it receives.

The Vanilla Deep neural network are not scalable for image classification as image size increases due to an exponential increase in number of weights.

In the MNIST dataset, the problem appears manageble with 28x28 Pixel images, but Scaling up to larger images rapidly increases the complixity.

for large image, such as 200x200 full-color images, The no of weights required for just the input layer becomes high, leading to high risk of overfitting and ineffiriency.

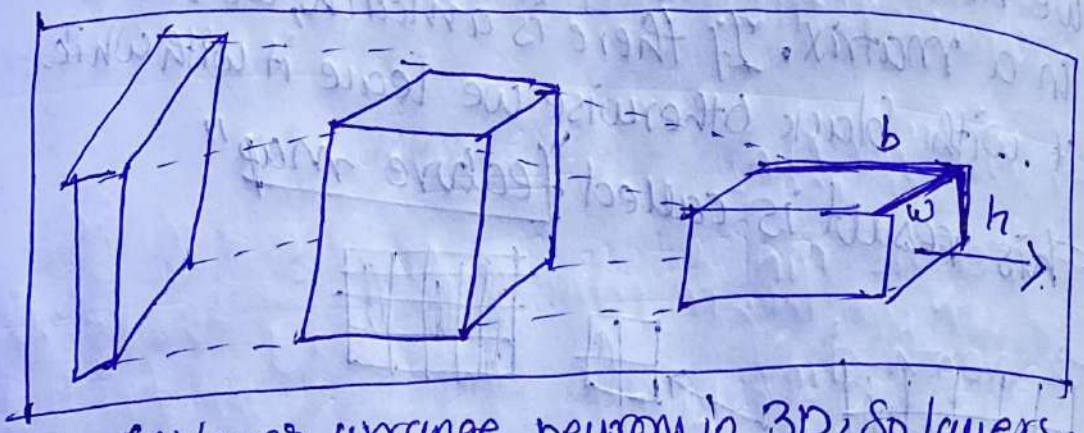
The CNN addres this Scalability issue by arranging neurons in 3D and connecting each neuron to only a Small, localized region of the previous layer.



This architecture mimics human visual processing and reduce the no of parameters significantly.

CNN manage the complexity by processing info in a 3D; So layers have width, length & depth; So new volumes of Procesed info layer by layer.

This design of CN layer avoids the problem of full connectivity and reduce the potential for overfitting while handling larger images more efficiently.



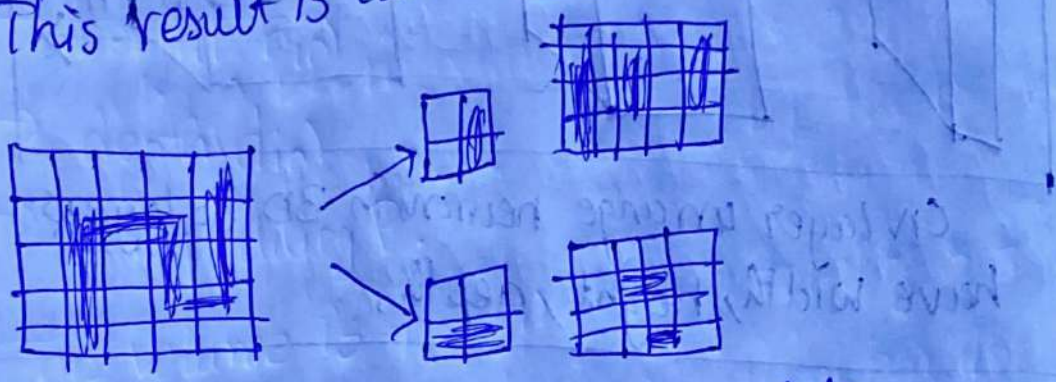
CN layer arrange neurons in 3D, so layers have width, height, depth.

Filters and Feature Map:

The visual cortex layers structures, where each layer builds upon the features detected by the previous one, inspired the design of CN layers in neural networks.

The filters is essentially a feature dector that slide acros the image to identify features like edges, lines & angles.

To detect Vertical lines, we would use the feature dectors on the top, Slide it across the image and at Each and Every step check if we have a match, we keep track of answer in a matrix. If there is a match, we shade it with black otherwise we leave it with white. This result is called "feature map"



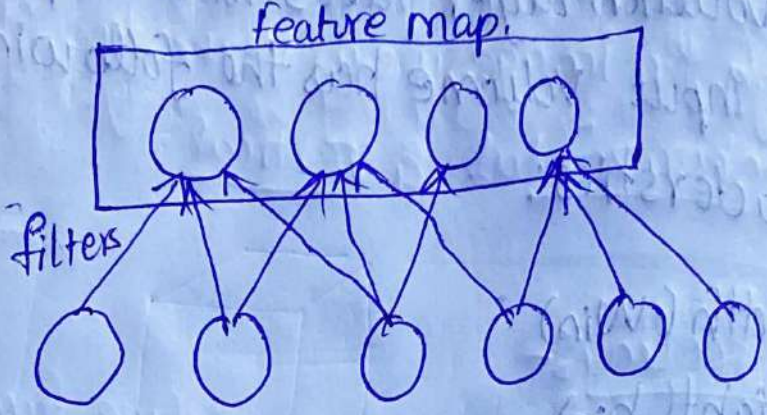
This operation is called Convolution.
A neuron in the feature map is activated if the filter contributes to the detection of a feature in the corresponding position in the previous layer

---

Expressing the Convolution Operation as neurons in a network

Layers of neurons in a feed forward neural net represents either the orginal

image or a feature map that get replicated across the Entirety of the input.



Let's denote the k^{th} feature map in layer m as m^k. $W \rightarrow$ corresponding filters weight. $b^k \rightarrow$ biaf. we mathimatically Expres feature map as:

$$m_{ij}^k = f\left((W * x)_{ij} + b^k\right)$$

The fundamental operation of Slides filters across the input data to produce features, maps, Capturing more Complex features. They capture Patterns like edges or textures.

filters in CNN have depth, allowing them to Process multi-channel input, such as Colour images.

The CNN Converts one Volume of values into another Volumes of values.

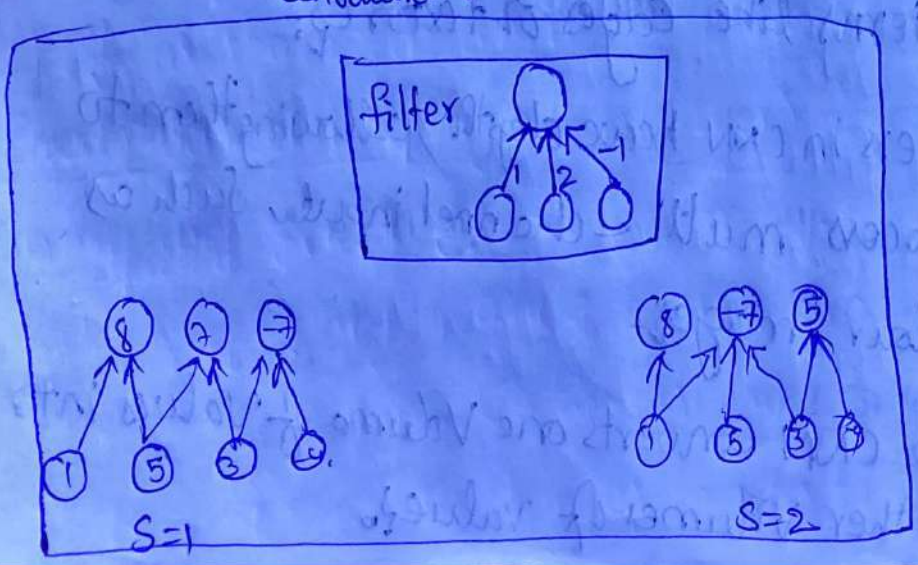
Full Description of Convolution layer

a Convolution layer takes in a input Volume, This input volume has the following Characterstics:

→ Width (Win)
→ height (hin)
→ depth (din)
→ Zero padding P.

This volume is processed by a total of K filters, This filters has many no of hyper parameters:

→ Spatial Extent e → which is equal to the filters height & width.
→ Stride S → distance b/w consective applications of the filters on the input volume.
→ bias b → which is added to each component of the Convolution.



The Result in the output volume has :

→ function f,

→ Width $W_{out} = \left[\frac{W_{in} - e + 2p}{S}\right] + 1$

→ height $h_{out} = \left[\frac{h_{in} - e + 2p}{S}\right] + 1$

→ depth $d_{out} = k$

w=5
h=5
d=3
Zero padd=1

w=3
h=3
d=2.

Padding :-

If the input is nxn and filter size is fxf the output size will be $(n-f+1) \times (n-f+1)$

There are two disadvantages:

→ Every time we apply a convolution Operation, the Image size shrinks

→ Pixels present in the corner of the images are used only a few number of times during convolutional as compared to center pixels.

To overcome the issue we can pad the image with additional borders. We add one pixel around the edges. This means input size will be 9x9 instead of 7x7.

input : nxn
Padding : p
filter size : fxf

output : $(n+2p-f+1) \times (n+2p-f+1)$

Striding

No of steps taken for convolution.

Suppose we choose a stride of 2. So while convoluting through the image, we will take two steps: both in horizontal & vertical

---

Max Pooling:-

It returns the maximum value from the portion of the image covered by the kernel.

~~We can describe~~

The pooling layer is responsible for Reducing the Spatial Size of the convolved feature.

This is to decrease the computational power required to process the data through dimensionality reduction and hence speed up the computation.

↓

two parameters:

1) Spatial Extent "e".

2) Stride "s".

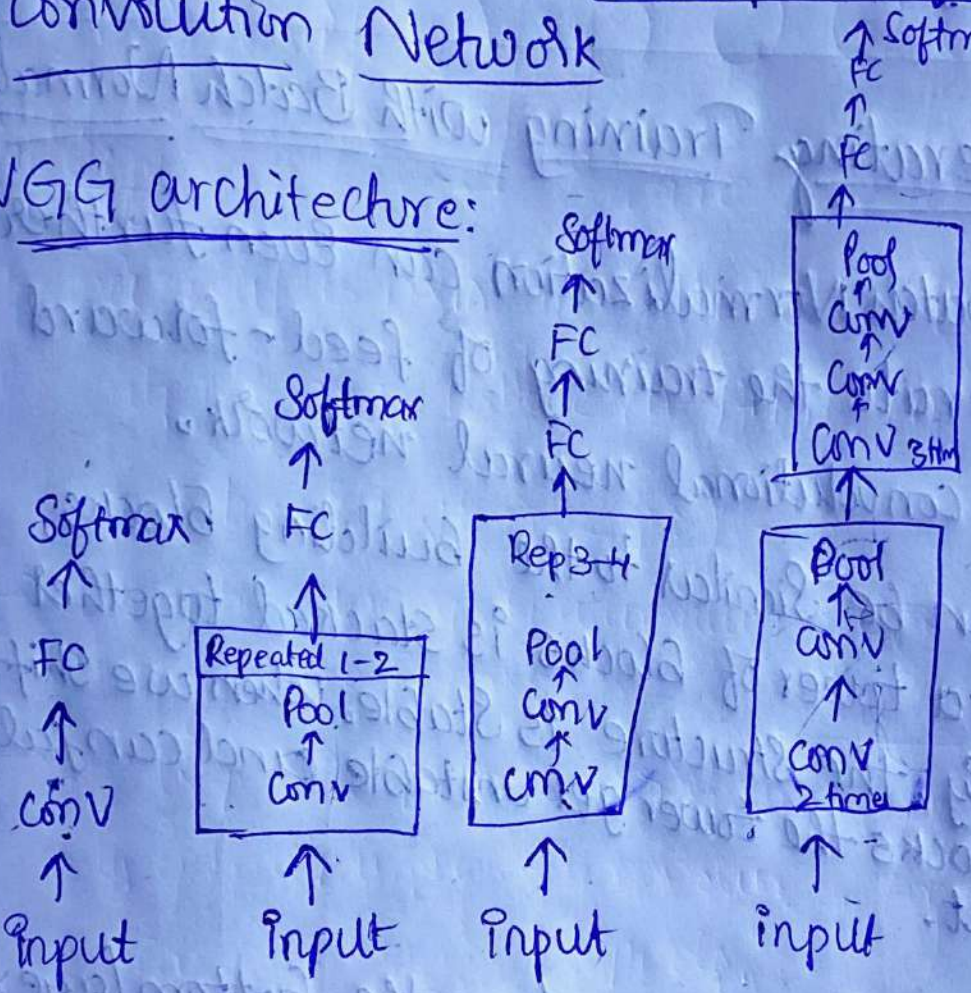
Width $W_{out} = \left[\frac{W_{in} - e}{s}\right] + 1$.

Height $H_{out} = \left[\frac{h_{in} - e}{s}\right] + 1$.

(1+V):

Full Architectural Description of Convolution Network

VGG architecture:



FC → Flatten

The VGG architecture is a deep convolution neural network known for its simplicity and its deep stack of convolution layers with

Small filters.

Input layer: The input layer would be size 28x28x3, assuming the images are RGB.

Link

Accelerating Training with Batch Normaliz

The batch Normalization can even further accelerate the training of feed-forward and convolutional neural network.

It can be similar to the building blocks, when a tower of blocks is stacked together neatly, the structure is stable. When we shift the blocks the tower get unstable and can fall a part.

A similar phenomeon where the bottom layer (first layer) and top layer (last layer) happens, when we give data to the bottom layer the data keep changing while passing from bottom layer to upper layers. So the last layer has to keep adapting to the changes in order to give

~~UNITOII~~

1) Collect output .So the more the layers we have the more problem we get.

Normalization of Image inputs helps out the training process by making it more robust to variations. Batch Normalization takes this a step further by normalizing inputs to every layer in our neural network.

We modify our network to include the below operations for batch Normalization'.

1) Grab the vectors of bias which is incoming to a layer before they pass through the nonlinearity.

2) Normalize each component of the vector of bias across all Examples of the minibatch ~~by Subtracting the~~.

3) Given Normalized input $\hat{x}$, use an affine transform to restore representation power with 2 Vectors of Parameter : $\gamma\hat{x} + \beta$.

Batch Normalization also allows us to Significantly ↑ the learning rate.

UNIT-II

Questions:

1) Principal component Analysis / Diff b/w PCA & Autoencoders.
2) Autoencoder Architechure.
3) Denoising of Autoencoder for
4) Sparsity in Autoencoder.
5) Lower-Dimensional Representation
6) Diff b/w AE & PCA.
7)
8)
9)
10)
11)
12)
13)
14)
15)

Chapter 2:-

1) Learning Lower Dimensional Representation

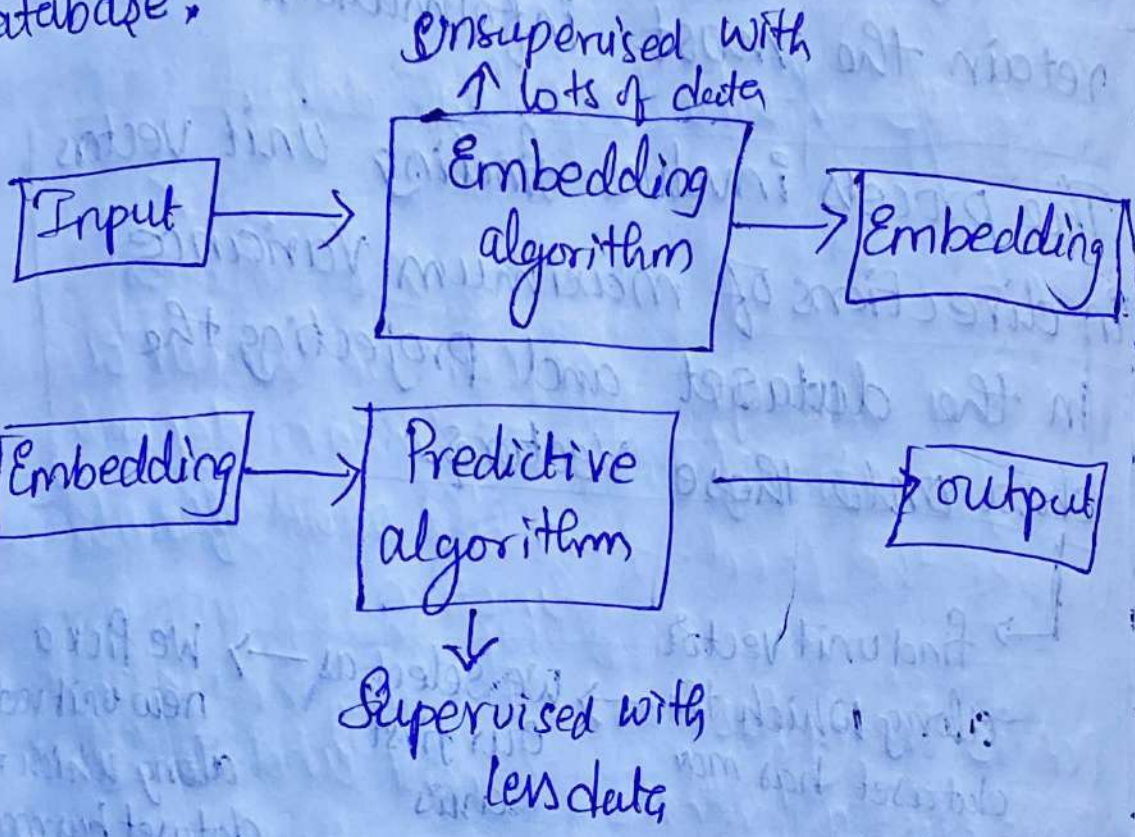
Large input vectors leads to larger models with many parameters, which can be Expressive but are also data-hungry and Prone to Overfitting.

The Convolutional architectures address the curses of dimensionality by Reducing the number of Parameters, maintaining Expressiveness without need for Extensive training data.

There is often a Scarcity of labeled data, which is necessary for training Convolutional networks, leading to the needs for methods that can work effectively with less labeled data.

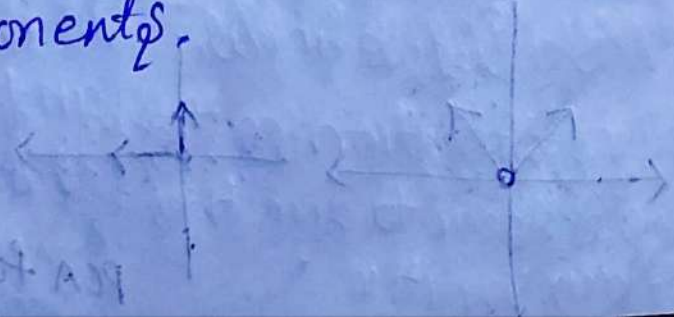
One Solution is to develop models that learn from unlabeled data by creating lower-dimensional embeddings in an unsupervised manner.

These Embeddings capture essential features and can be used with smaller models, reducing the requirments for large labeled database.



Principal Component Analysis:

Principal component Analysis is a Statistical Procedure that aims to transform a set of Possibly correlated variables into a set of linearly uncorrelated variables known as principal components.



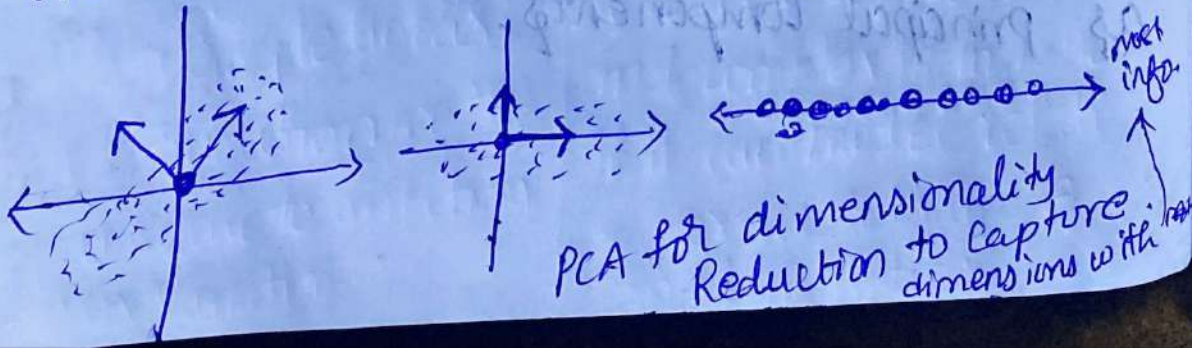
The goal of PCA ~~is to retain as much~~ seeks to reduce the dimensionality of data by finding the new axes that retain the most information.

The process involves finding unit vectors in directions of maximum variance in the dataset and projecting the data onto these vectors.

→ find unit vector along which the dataset has max variance → We select as our first axis → We pick a new unit vector along which the dataset has max variance. ← This is our second axis

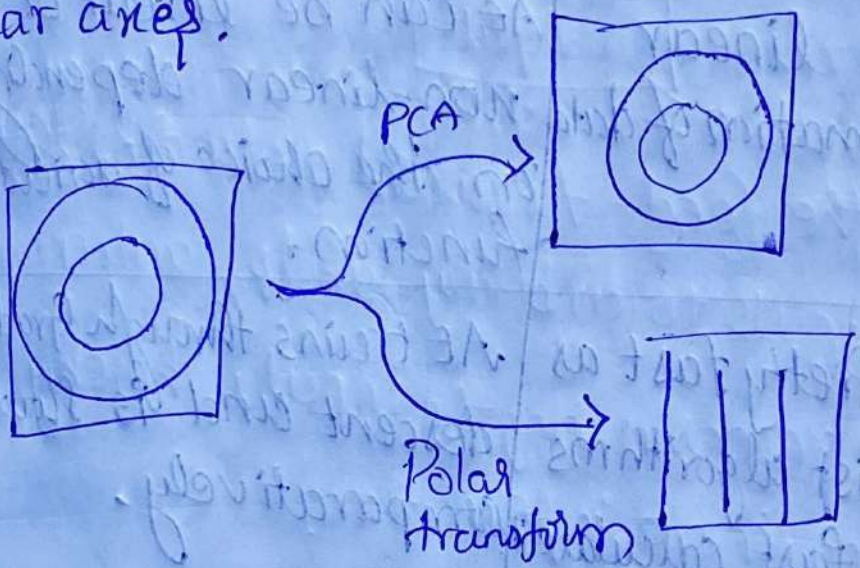
Continue this process until we have found a total of d new vectors. ← Project our data onto this new set of axis.

This is result is shown in below fig.



The dataset is represented as a matrix, and an Embedding matrix is created that Reduces dimensions while preserving as much info as possible.

PCA can fail With complex datasets, Particularly where important patterns are nonlinear, as it primarily capture Variance along linear axes.



A Situation in which PCA fails to optimally transform the data for dimensionality Reducing

The nonlinear Relationships in data, such as the above concentric circles, may Require diff approches, like polar transformation, to Separate features Effectively.

The limitations of PCA in dealing with nonlinear relationships highlight the needs

for non-linear dimensionality Reduction techniques, which are addressed by Neural models in deep learning.

PCA v/s Auto Encoders

PCA	AE
PCA is a linear transformation of data	AE can be linear or non-linear depending on the choice of activation function.
PCA is Pretty fast as there Exist algorithms that can fast calculate it	AE trains through Gradient descent and is slower comparatively.
PCA Projects data into dimensions that are orthogonal to each other resulting in very low or close to zero correlation in the Projected data.	AE transformed data doesn't guarantee that because the way it's trained is merely to minimize the reconstruction loss.
PCA is a simple linear transformation on the input space to directions of max variations	AE is more sophisticated and complex technique that can model relatively complex relationships & non-linearities.

one rule to thumb could be the size of Data. PCA for small datasets	AE for larger dataset.
PCA hyperparameter is "K"	AE it is the architecture of the neural network.
AE with single layer & linear activation has similar performance as PCA.	AE with multiple layer, and non-activation function termed as Deep Autoencoder Prone to Overfitting.

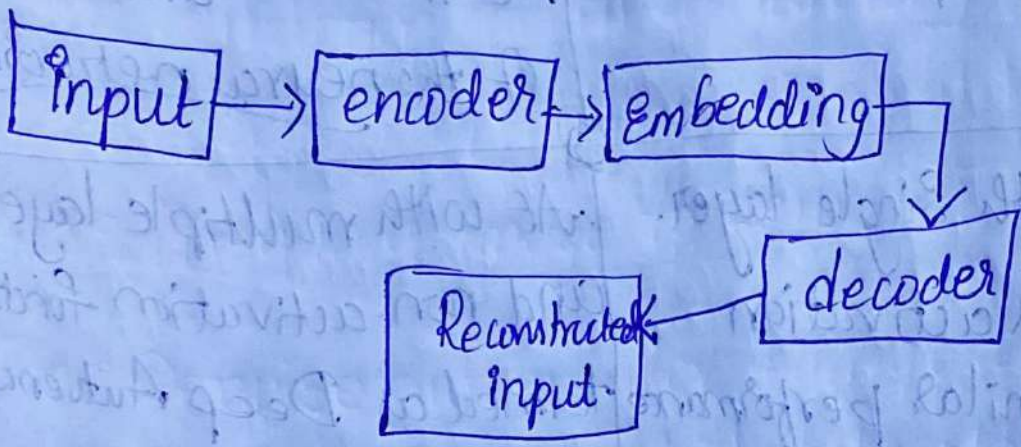
Autoencoder Architecture:

The feed-forward networks learn increasingly refined representation of input data, with the final convolutional layer output serving as a compressed representation.

Autoencoders are introduced as an architecture designed to address the loss of information by capturing and compressing input data into a lower-dimensional space.

An autoencoder consist of two main components:

an encoder the compress the input into a code, and a decoder the reconstructs the original input from this code.

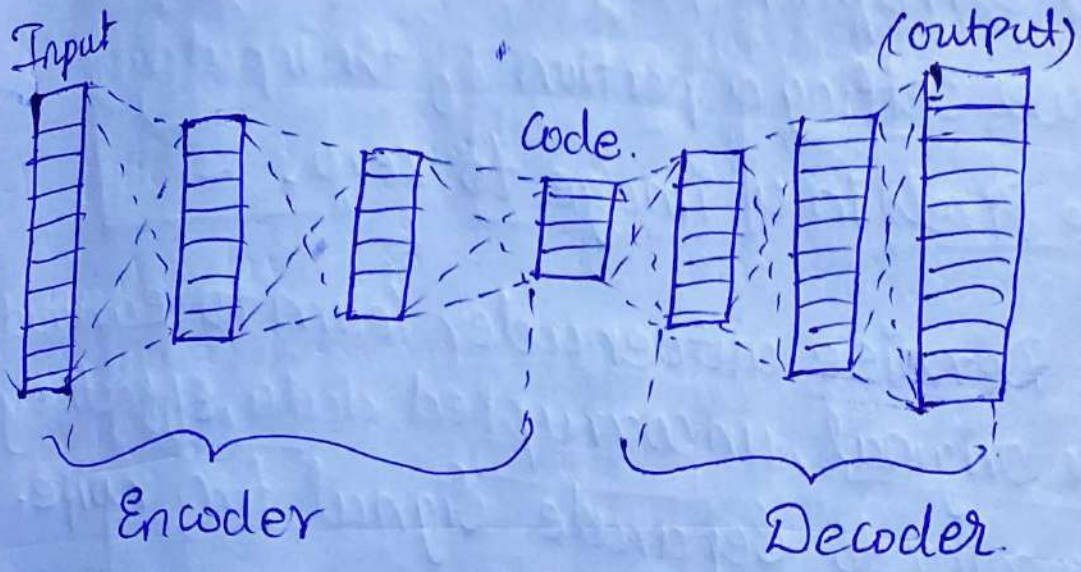


Unlike the feed-forward networks that map inputs to arbitary labels, autoencoder focus on reconstructing the input, thereby retaining more information.

The encoder part of the autoencoder learns to identify and encode the most critical features of the input data.

The decoder part learns to Revere the encoding process to reconstruct the input data from the encoded representation

Autoencoders can be used to generate lower dimensional data Representation without the supervision required in traditional methods.



Denoising of autoencoder

Denoising:

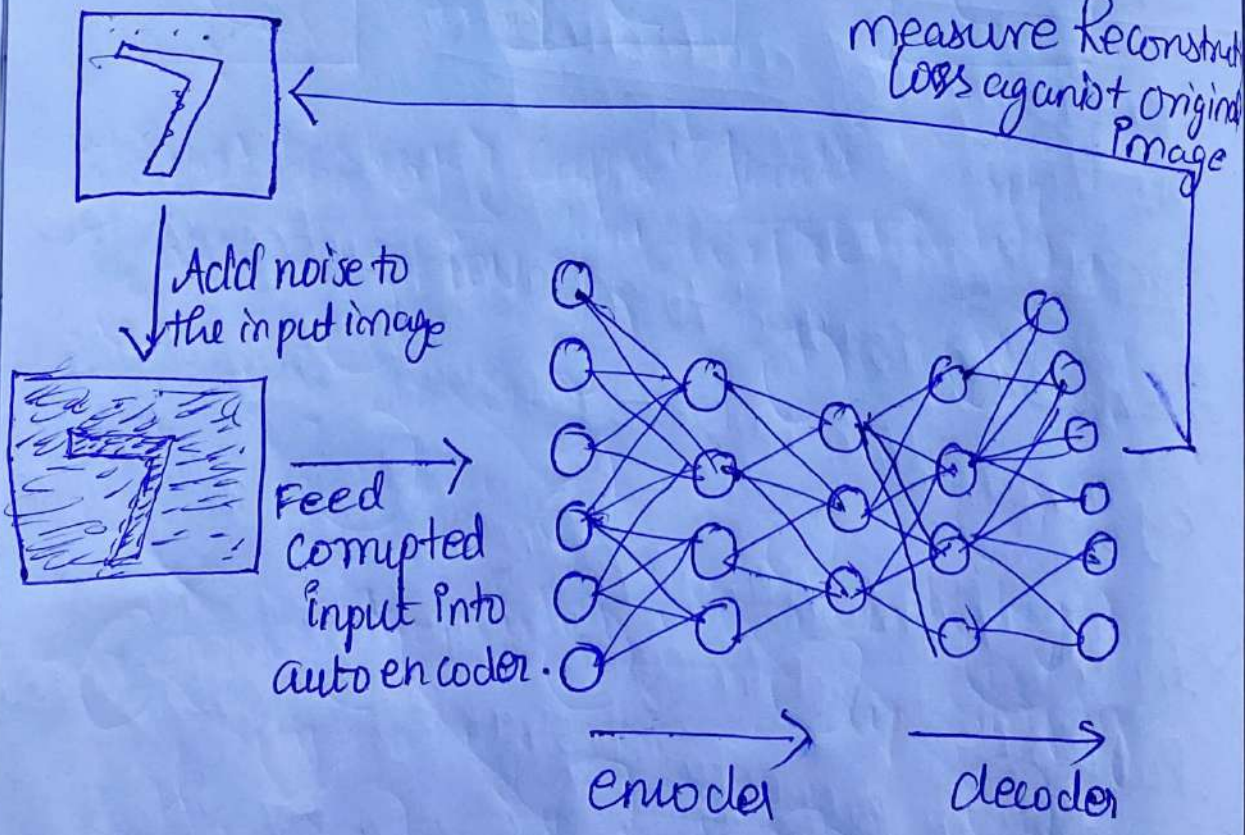
Denoising is used to enhance autoencoders by generating noise-resistant Embedding.

In the human Perception we can recognize Patterns and images despite significant noise, a property desirable for ML models we can even distinguish 8 and 7 if all the pixels got corrupted easily.

The Denoising encoders are trained using corrupted inputs, forcing them to learn robust features.

The corruption process in denoising involves setting a portion of the input data, like Pixels in a image to zero.

The Denoise autoencoder learn to reconstruct the original, uncorrupted data, effectively learning to separate signal from noise.



The geometric interpretation involves the concept of a manifold, the shape within the data that the autoencoder learns to reconstruct.

An autoencoder must identify whether a data point belongs to one manifold or another during the reconstruction process.

The denoising process conceptually expands the data space to include points around the manifold, with the autoencoder then collapsing these points back onto the manifold.

The denoising objective helps the autoencoder learn to distinguish b/w generalizable features and noise.

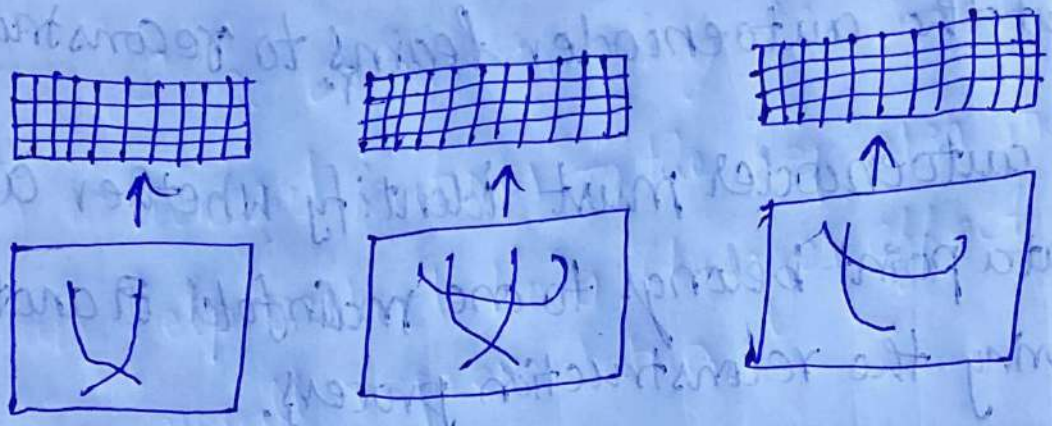
Original Image 

Corrupted Image 

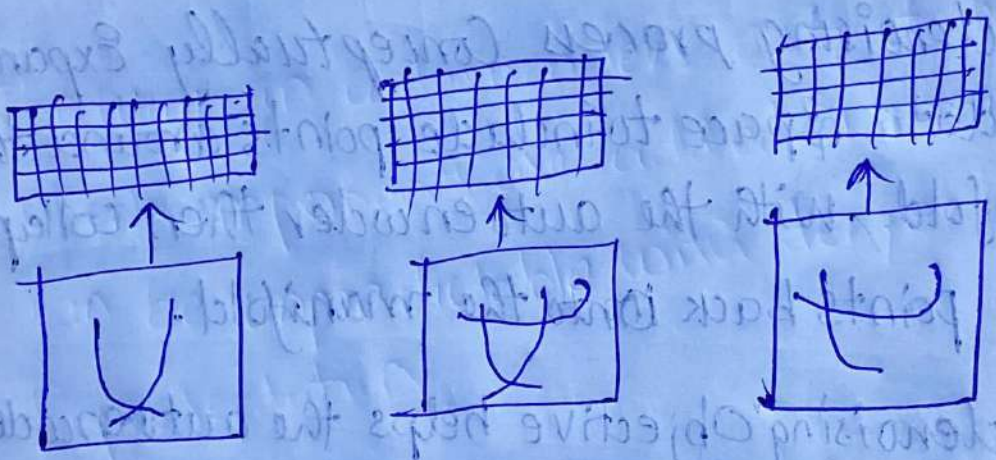
Reconstructed Image 

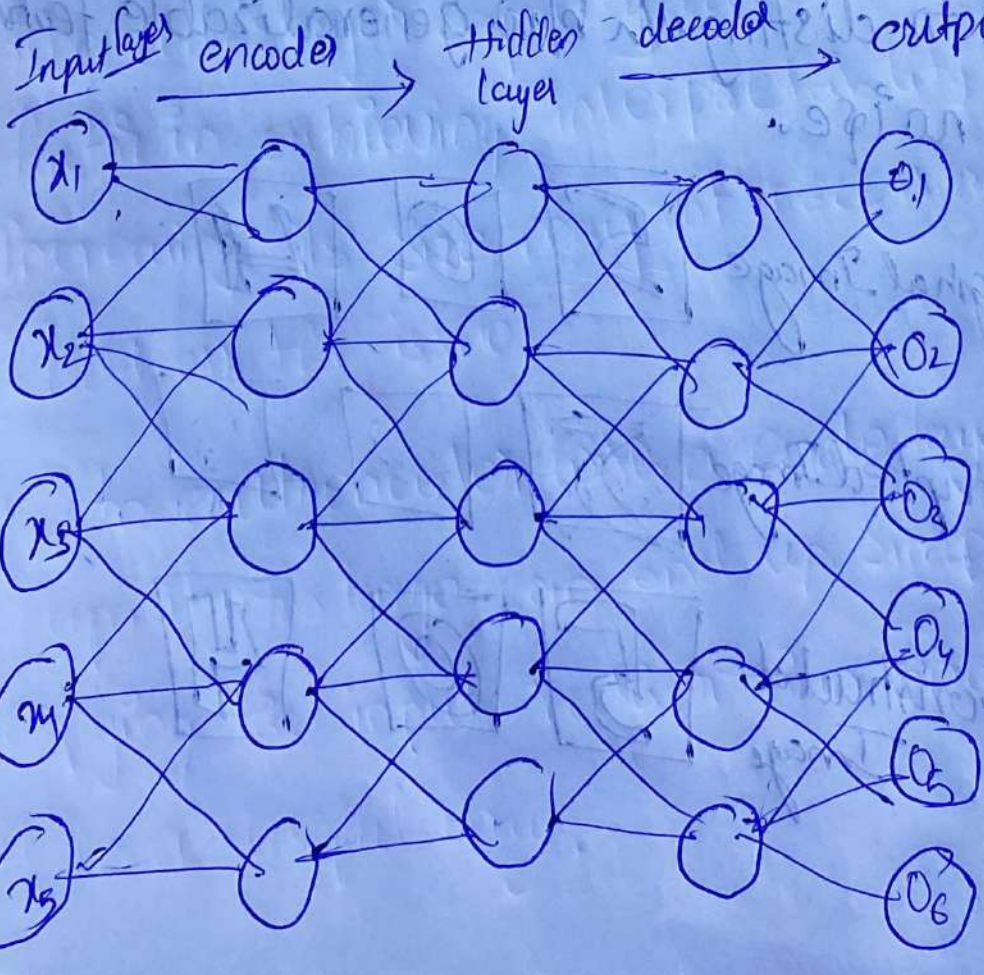
Sparsity in Autoencoders:

A



B





Deep learning models struggle with interpretability due to their complexity and non linearity. Deep learning models are generally very difficult to interpert because of the non linearities and massive number of paleameters that make up a model.

The Interpereta bility can be solved by auto-encoder. In general, an autoencoder Representations are dense, which means that many features are combined in ways that are diffcult to separate or interpret.

In figA. The activation of a dense representation combine and overlay information from multiple features in ways that are diffcult to interpret.

Introducing Sparsity into autoencoder representation can help make the model output more interpretable.

The goal is to achieve a one-to-one corresponding b/w the high level features in the input

data and the components in the autoencoder code.

→ The Fig B. colour-coding the contributions of individual features to the Representation can help understand how changes effect the overall representation.

→ The way an autoencoders output changes when components are added or removed can be unpredictable with dense Representation.

→ The code's layer capacity can be limiting factor in achieving interpretability, increasing its size may not necessarily address the issue.

→ An autoencoder with a code layer capacity too large can end up simply copying the input rather than encoding it in a useful way.

→ The Autoencoder can effectively summerize data points when all relevant info is present in the individual data points themselves.