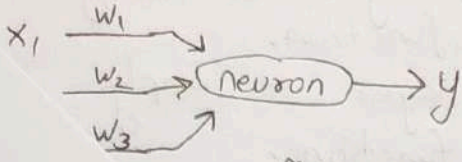
28/1/25

ANN



$x_i \rightarrow$ i/p
$w_i \rightarrow$ weights
$y \rightarrow$ o/p

$$\sum_{i=1}^{m} w_i x_i = f(x)$$

Apply activation function ϕ, $\phi(f(x)) = y$

$$\phi(f(x)) = \phi\left[\sum_{i=1}^{m} (w_i x_i)\right] = y$$

s that determines strength of the connection b/w
resent layers. It is a numerical value
between two neurons.
layers of neurons: i/p layer,

tical function used in a neural
to the model. Without
ke a linear model which
complex patterns in

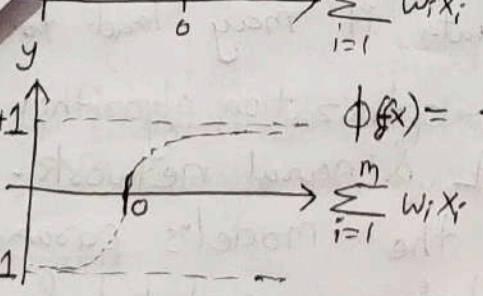
Deep Learning

x
≥ 0
$x < 0$

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

$$\sum_{i=1}^{m} w_i x_i$$

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



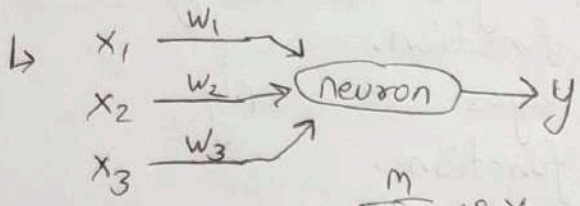
$$\phi(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$$

Binary variables (logistic Regression) threshold activation function & Sigmoid activation functions are used.

2025/4/27 10:48

28/1/25

ANN



$x_i \rightarrow$ i/p
$w_i \rightarrow$ weights
$y \rightarrow$ o/p

step 1: $\sum_{i=1}^{m} w_i x_i = f(x)$

step 2: Apply activation function ϕ, $\phi(f(x)) = y$

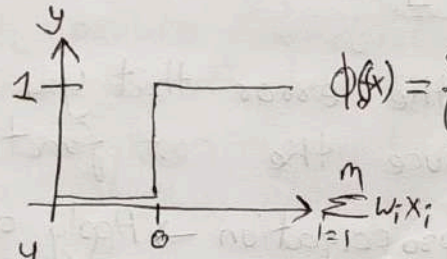
$$\phi(f(x)) = \phi\left[\sum_{i=1}^{m}(w_i x_i)\right] = y$$

↳ Weights are the parameters that determines strength of the connection b/w 2 neurons (or nodes) in different layers. It is a numerical value assigned to the connection between two neurons.

↳ A neural network consists of layers of neurons: i/p layer, hidden layers, o/p layers.

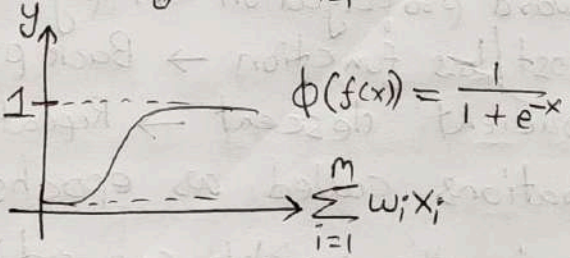
↳ Activation function is a mathematical function used in a neural network to introduce non linearity into the model. Without activation function, network behaves like a linear model which would not be able to learn & represent complex patterns in data.

Threshold function:

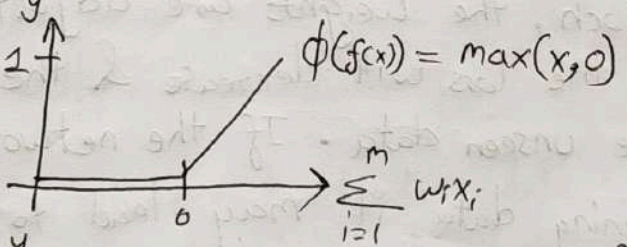


$$\phi(f(x)) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

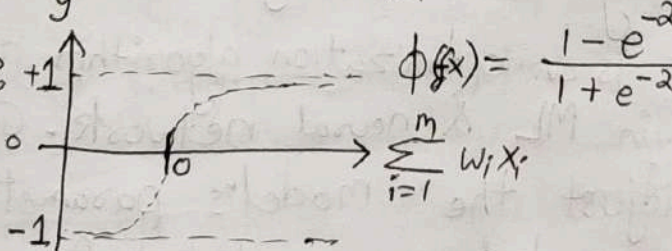
Sigmoid function:



Rectifier function:

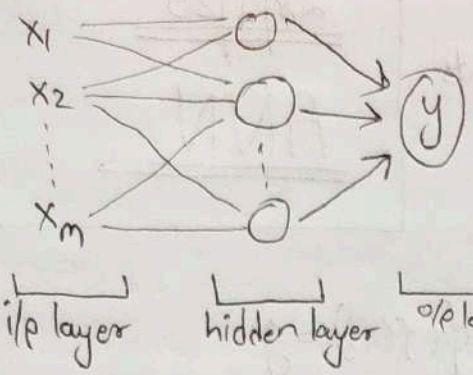


Hyperbolic Tangent (Tanh):



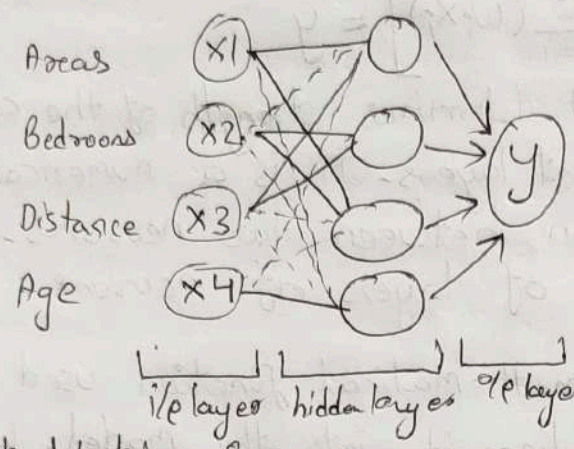
↳ For Binary variables (logistic Regression) threshold activation function & Sigmoid activation functions are used.

2025/4/27 10:49

↳ 

- In hidden layer we apply Rectifier function.
- In o/p layer we apply Sigmoid function.

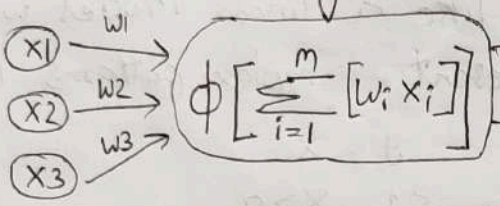
↳ Example :



$\Rightarrow Price = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4$

$Price = \sum_{i=1}^{4} w_i x_i$

Neural Network Working :-



$\hat{y}$ (Predicted %)

y (Actual %)

so, Cost function (C) :

$C = \frac{(\hat{y} - y)^2}{2}$

↳ Cost function tells the error that you have in your prediction. Our goal is to reduce the cost function.

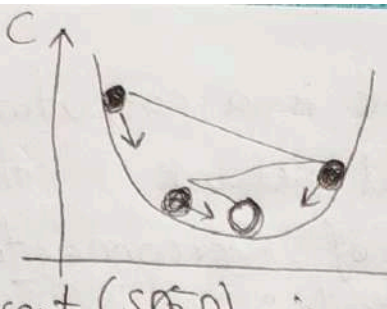
↳ Give i/p → Forward propagation → Apply activation function → → Calculate cost/loss function → Back propagation → Update weights by gradient descent → Repeat these many steps for many iterations called as epochs.

During each epoch, the weights are adjusted to reduce the loss. Over time, the loss will decrease & the network will generalize the unseen data. If the network learns too much from the training data it may lead to overfitting.

↳ Gradient descent is an optimization algorithm used to minimize the loss function in ML & neural network. Goal of gradient descent is to adjust the models parameters in order to minimize the error between predicted & actual values.

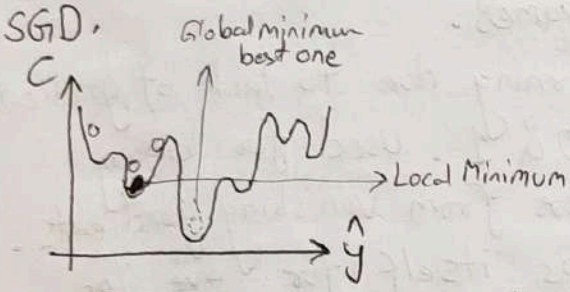
2025/4/27 10:49

↳ Gradent descent :



$$C = \frac{(\hat{y}-y)^2}{2}$$

↳ Stochastic Gradient Descent (SGD) is a variant of Gradient descent algorithm that is used for optimizing ML models. It address the computational inefficiency of traditional gradient descent methods when dealing with large datasets in ML projects. In SGD, only a single random training sample is selected to calculate the gradient & update the model parameters. Computational efficiency is the advantage of using SGD.



↳ Gradien descent : Uses entire training dataset to calculate gradient & update model's parameters (weight). Computes avg.. gradient for all data points & takes one step to update weights. It can be slower for large datasets. It has smoother convergence since it uses full datasets.

Stochastic Gradient descent : It uses one random training sample/example at a time to calculate gradient & update weight (model params). It computes gradient for one data point & updates the weight. It is fuster bcz it updates weights more frequently. It can be noisy & fluctuate alot.

Gradient descent uses whole dataset to update weights. It is more accurate but slower. Stochastic Gradient descent uses one example at a time to update weights. It is faster & it can be noisy & fluctuates more. For large datasets SGD is prefered due to its speed while GD can be more stable but slower.

ANN

↳ Artificial Neural Network is a computational model inspired by how biological neural network in the brain process the information. It consists of interconnected layers of nodes or neurons. Its primary goal is to learn patterns & relationships from data by adjusting weights between the neurons during training process.

↳ Key components :- i/p layer, hidden layers, o/p layers, Neurons (Nodes), weights & Biases, Activation function

↳ Working :- Forwad Propagation & Back Propagation

↳ ANNs can be used in Image recognition, speech recognition, NLP Natural Language Processing, playing games.

↳ Threshold function : Not often used in deep learning due to lack of gradients.

Sigmoid function : Smooth, o/p values b/w 0 & 1. Used for binary classification but suffers from vanishing gradients.

Relu function : o/p 0 for -ve i/ps. i/ps itself for +ve i/ps. Fast convergence but has dying ReLU problems.

Tanh : Like sigmoid but zero-centered. o/p b/w -1 & +1. Helps with training, though still suffers vanishing gradients.

2025/4/27 10:50

03/02/2025

CNN

CNN

↳ It is Convolutional Neural Network. It is a type of neural network particularly effective for processing & analyzing visual data such as images or video.

CNNs inspired by the structure of the visual ~~context~~ cortex & are designed to automatically and adaptively learn spatial hierarchyes of features from images.

↳ key components : Convolutional layers, Activation functions, Pooling layer, Fully Connected layers, Output layers.

↳ CNNs are successfully in computer vision tasks like image classification, object detection, Segmentation due to their ability to learn complex patterns in image data.

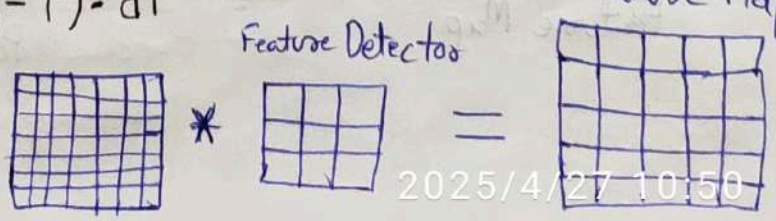
↳ Working : i/p image → Convolution Layer (Feature extraction) → Activation Function (ReLU)
Multiple Convolution + Pooling Layers ← Pooling Layers (DownSampling) ↵
↳ Flattening → Fully Connected Layers (Dense Layers) → o/p layer
Back Propagation ↵

Convolution – Extracts low level features
Relu Activation – Add non-Linearity
Pooling – Reduces size of feature map
Multiple Convolution + Pooling Layers – Learns more complex features at higher levels
Flattening – Converts features maps into a 1D vector
Fully Connected Layer – Combine the features and make predictions
output layer – Produces final results.

↳ Geoffrey Hinto is the godfather of Artificial Neural Network & Deep Learning (ANN & DL). Yann LeCun is the grandfather of CNN.

↳ Convolution is core operation in CNN & is essential for extractive features from i/p data such as images. The result of the convolution is the feature map.

$$(f * g)(t) \overset{\text{def}}{=} \int_{-\alpha}^{\alpha} f(\tau)\, g(t-\tau)\, d\tau$$



2025/4/27 10:50

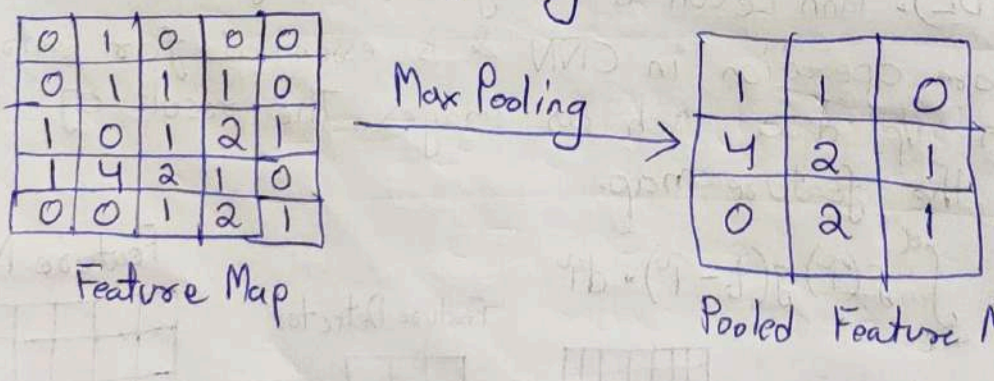
↳ ReLU is also known as Rectified Linear Unit & it is most commonly used activation functions in deep learning (DL) espicially in CNNs. It introduces non-linearity into the model, allowing to learn more complex patterns. $f(x) = \max(0, x)$

- It introduces non linearity into the model. It is computationally efficienct bcz its easy to implement & requires only a comparision operation. It helps to avoid vanishing gradient problem that can occour with other activation functions, espicially in Deep networks. It activates only some neurons which introduces sparcity into the network which can help effecient computation & reduce the chance of overfitting. Only subset of neurons contribute to output any time.

↳ Pooling is an operation in CNN. It helps in reducing spatial dimen-sions of feature map which makes the network more computat-ionally effcient and less prone to overfitting. It contributes to translational invarience of the model which means the network can recognize objects even if they are slightly changed. i.e. shifted or distorted.

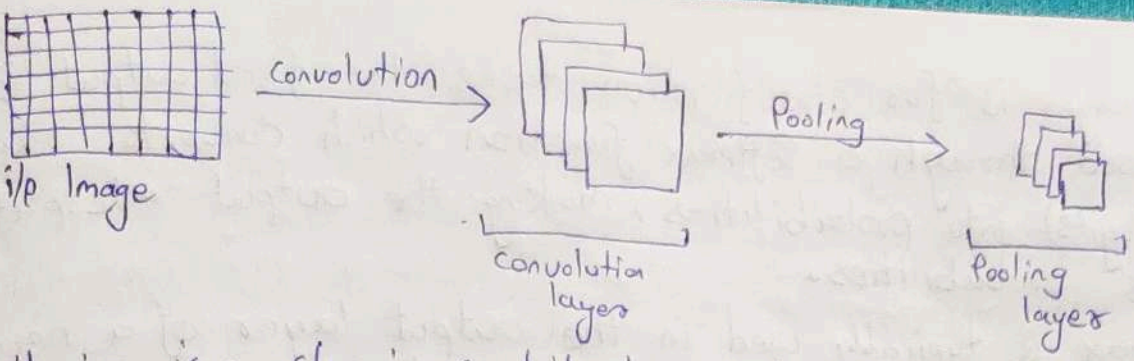
Types: Max Pooling — Maximum value in window is selected
Avg Pooling — Average value within the filter window is calculated
Global Pooling — Reduces the entire featuremap to a single value.

Benefits: Dimensionality reduction, Translational Invariance, Prevents overfitting

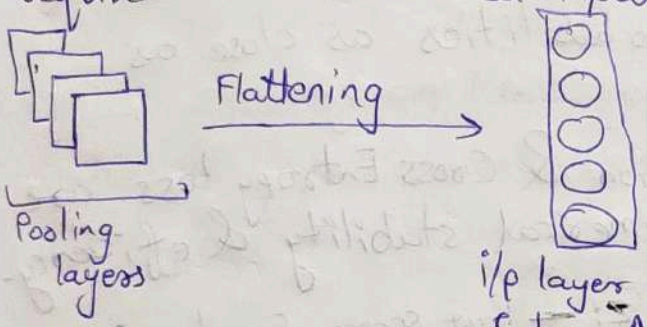


Feature Map
Pooled Feature Map

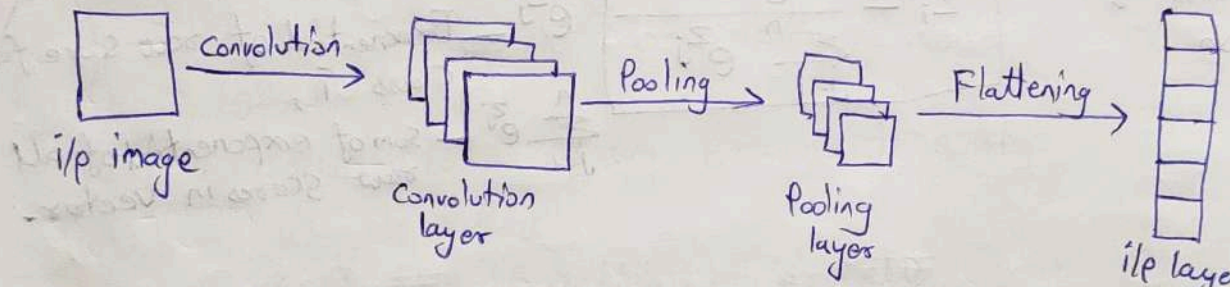
2025/4/27 10:50



↳ Flattening is a step in architecture of a Convolutional Neural Network CNN, particularly when transitioning from the convolutional and Pooling layers to the fully connected layers. It is a process of converting multi dimensional tensor into a one dimensional vector, so that it can be passed into fully connected layers which required one dimensional input.



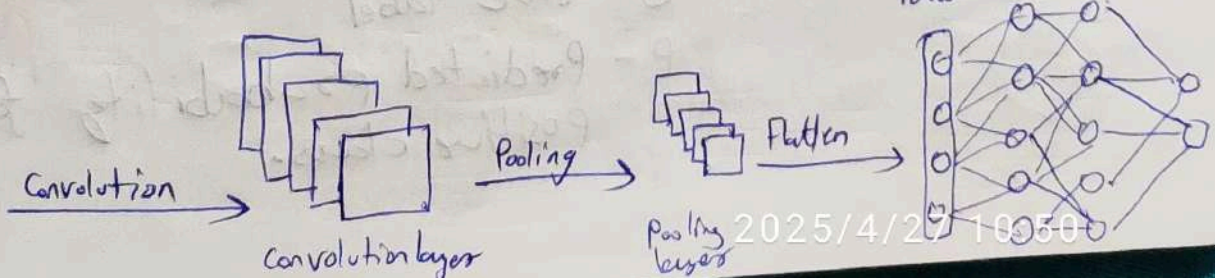
So,



↳ Full connection is a Fully Connected Layer (FC layer) which is also known as Dense layer, it is a type of layer where each neuron is connected to every neuron in the previous layer. All neurons from the previous layer are fully connected to each neuron in the fully connected layer.

Structure : i/p → Weights → Biases → Activation Function → Output

Applications : classification & Regression



↳ Softmax Layer is for classification tasks, the final output is often passed through a Softmax function which converts raw scores (logits) into probabilities, making the output interpretable as class probabilities.

The softmax is typically used in the output layer of a neural network when dealing with multiclass classification problems. Each output value will be between 0 and 1. Sum of all output probabilities will be equal to 1.

↳ Cross Entropy is a loss function commonly used for classification problems particularly when using Softmax. It measures between the true class labels & the predicted probabilities as close as possible to the actual labels.

In DL frameworks the softmax function & Cross Entropy loss are combined into one function for numerical stability & efficiency.

↳ Softmax function :

$$Z_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

z_i — Raw score for class i,

e^{z_j} — Exponential of raw score for class i,

$\sum_{j=1}^{n} e^{z_j}$ — sum of exponential of all raw scores in vector.

Cross Entropy : For multiclass classification

$$L = -\sum_{i=1}^{N} y_i \cdot \log(p_i)$$

N - no. of classes

y_i - True label for class i

p_i - Predicted probability for class i

2025/4/27 10:51

For Binary classification

$$L = -[y \cdot \log(p) + (1-y) \cdot \log(1-p)]$$

y - true label

p - Predicted probability for the positive class.

9/2/25 RNN

RNN

↳ Recurrent Neural Network is a type of ANN designed for sequential data or time series data such as speech, text or video. RNNs have memory & can retain information from previous time steps.

↳ Each input is processed independently, the output of a neuron is fed back into network as input for next time step. This feedback loop allows the network to maintain a memory of previous input. Here, Output from previous step becomes part of the input for the next time step & allows the RNN to model temporal dependencies in data.

↳ Hidden state of an RNN represents the memory of the network. Each time step, network updates this state based on the current input & the previous hidden state.

$$h_t = f(W \cdot x_t + U \cdot h_{t-1} + b)$$

h_t - Hidden state at time step t.
X_t - Input at time t.
h_{t-1} - Hidden state from previous time step.
W, U, b - Weights & Bias
f - Activation function (tanh or ReLU)

↳ RNN are prone to a problem called vanishing Gradients, where the gradient (the signal used to adjust the weights) becomes too small as it propagates backward through many time steps. which makes it difficult for network to learn long-term dependencies. Exploding gradients can cause the gradients to become too large, leading to unstable training.

↳ To mitigate these issues architectures like LSTMs & GRUs were developed.

LSTMs (Long Short Term Memorys)
GRUs (Gated Recurrent Units)

2025/4/27 10:51

↳ LSTM (Long Short Term Memory) is a type of RNN design to handle long range dependencies better by introducing a more complex cell. The memory cell has gates that control the flow of information allowing the network to forget or remember information at each time step.

Forget Gate : Decides what information from previous state shoud be discarded.

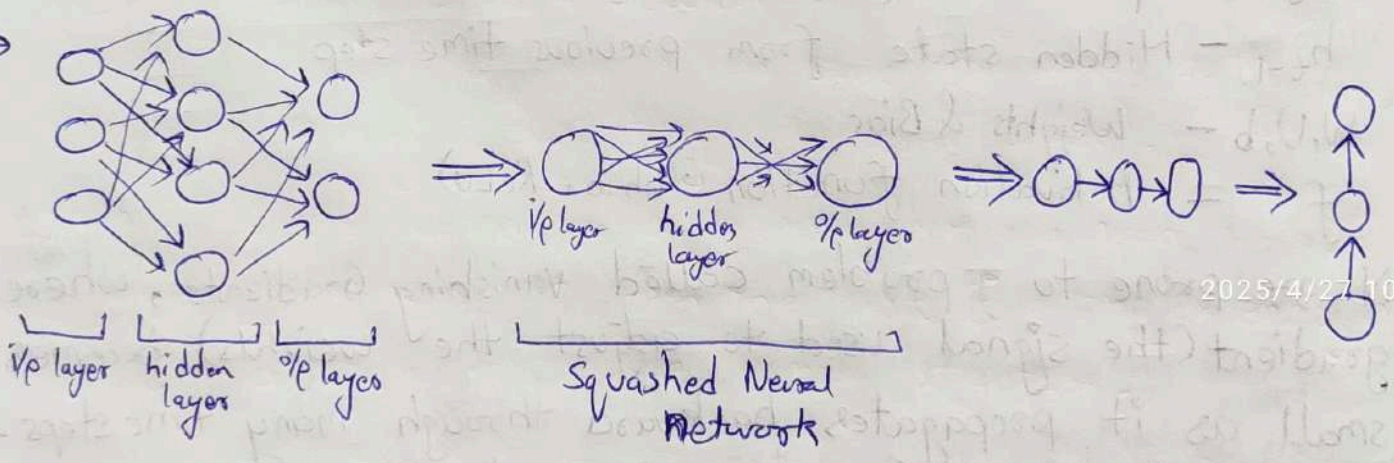
Input Gate : Controls what information should be added to the memory.

Output Gate : Determines what the next hidden state should be.

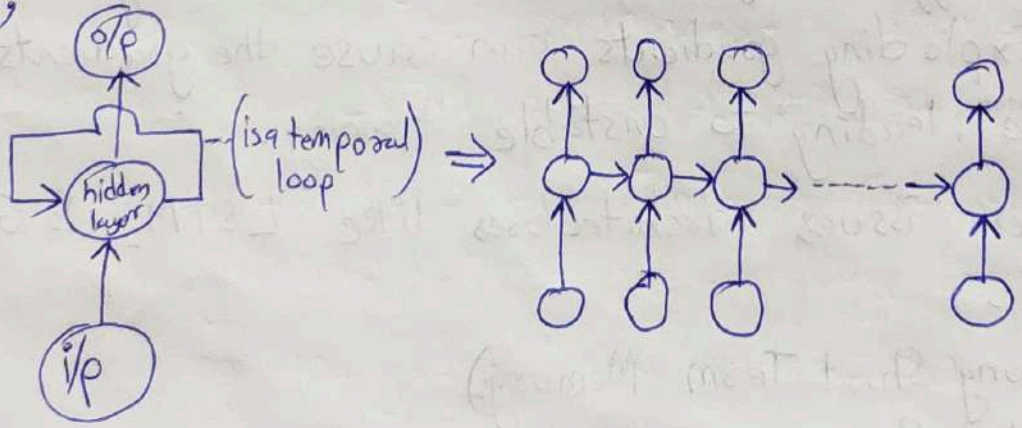
↳ Working : Forward pass, backpropagation through time (BPT)
Sequence to Sequence (Many to One, One to Many, Many to Many, Many to Many with diff.. lengths)

Applications : NLP (Natural Language Processing), Time Series predictions, speech recognition, Video analysis

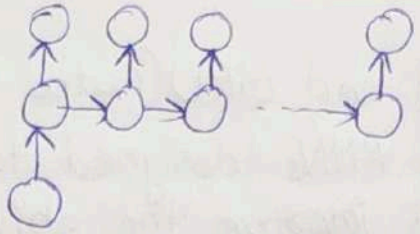
↳



So,



↳ One to Many :

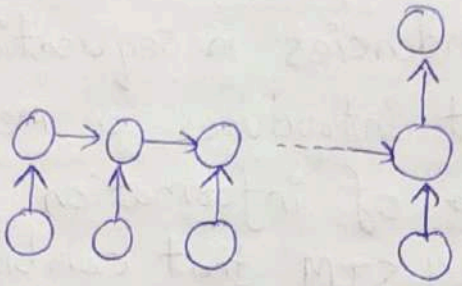


$W_{rec} \sim$ small → Vanishing

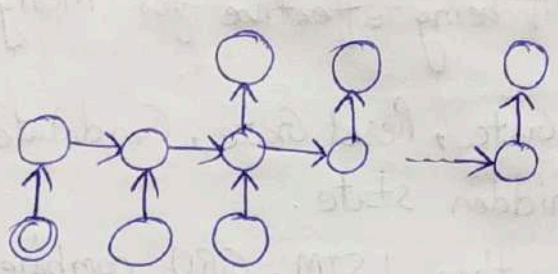
$W_{rec} \sim$ large → Exploding

One

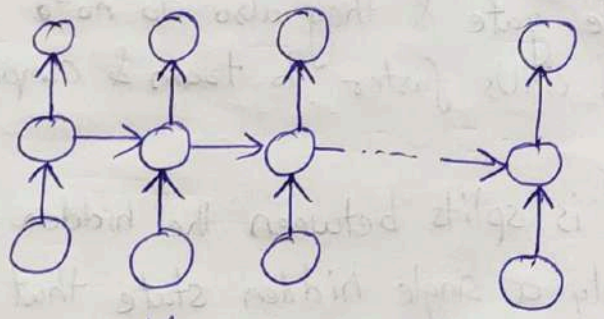
Many to ~~Many~~ :



Many to Many :



Many to Many :



↳ Vanishing gradient problem is a critical issue encountered during training of DL networks, particularly using traditional RNNs & deep feedforward networks. It occours when gradients become very small, effectively preventing the network from learning.

↳ ReLu & its variants : using activation functions to avoid small gradients.

LSTMs/GRUs : Use LSTM or GRU for sequential data tasks to handle long term dependencies.

Gradient clipping : Clip gradients during backpropagation to prevent extreme values.

Proper Initialization : Using techniques like Xavier or He initializations to prevent vanishing gradients.

Batch Normalization : Normalize activations to improve learning stability.

↳ LSTM (Long Short Term Memory) and GRU (Gated recurrent Unit) are both advanced versions of RNNs designed to address the vanishing gradient problem & improve the ability of RNNs to capture long term dependencies in sequential data.
It is a type of RNN that introduces a memory cell & gates to control the flow of information.

↳ GRU is a simplified version of LSTM that combines some of the gates and has fewer parameters, making it computationally more efficient while still being effective for many sequential tasks.

Key components : Update Gate, Reset Gate, Candidate hidden state, Final hidden state.

↳ GRU have lesser gates than LSTM. GRU combines forget & i/p gates into a single update gate & they also do not have a separate cell state. Which makes GRUs faster to train & computationally more efficient.
In LSTM memory is splits between the hidden state & cell state.
In GRU there is only a single hidden state that serves both purposes storing the memory & outputting the current state).

↳ LSTM is better suited for problems where you need to capture long range dependencies or when the task is complex enough to require a more expressive memory structure.
GRU is computationally more efficient & works well for many tasks with simpler long range dependencies. It often similar or better performance than LSTM in a lot of practical applications.

↳ Applications of LSTM & GRU : TimeSeries forecasting, Speech recognition, NLP (Natural Language Processing), Video analysis; Music generation.

2025/4/27 10:59

↳ Feature Scaling: Process of standardizing or normalizing the input features so that they have a consistent range or distribution. It is important preprocessing step bcz it can significantly effect performance & convergence speed of a Neural Network.

$$\text{standardisation } (X_{stand}) = \frac{X - \text{mean}(X)}{\text{std\_Dev}(X)}$$

$$\text{Normalisation } (X_{Norm}) = \frac{X - \min(X)}{\max(X) - \min(X)}$$

It is recommended that to use Normalisation for the output layer when using a sigmoid activation function in Deep Learning to ensure that the input data is in a suitable range for the model to perform ~~directly~~ efficiently.

2025/4/27 10:59

10/2/25 SOM

SOM (Self Organising Maps) :

↳ It is a type of unsupervised Learning Algorithm belongs to a class of Neural Networks. It is used for clustering & dimensionality reduction making them one of the tools in unsupervised learning.

↳ It is a type of ANN that is trained using Unsupervised Learning. It maps high dimensional data into a lower dimensional data/grid typically 2D to visualize data's inherit structure.

↳ It consists of grid of neurons, each with its own weight vector. The goal is to map the input data onto this grid in such a way that similar inputs activates nearby neurons. It produces a topological map, where similar data points are represented near each other on grid.

↳ Working :

Initialization – Neuron in map randomly initialises with weight vectors. Weight vector has same dimension as input data.

Competetion – Each i/p vector identifies Best Matching Unit (BMU), which neuron's weight vector is closest to the input vector.

Cooperation – Once BMU is found neighbour neurons in grid also updated.

Adaption – Weights of BMU & its neighbours are updated to become more like input vector. Learning rate & size of neighbourhood gradually decreases over time, that help refine map & allows it to convergence.

Convergence – After multiple iterations, weight vectors & convergence & map becomes organized such that similar input vectors are grouped together.

↳ Applications : Data Visualization, Clustering, Anomaly detection, Dimensionality Reduction, Feature Extraction.

Advantages : Non Linearity dimensionality reduction, Preservation of topological structure, Visualization.

↳ SOMs helps in organising & visualizige high dimensional data. Particularly effective for clustering, anomaly detection, dimensionality reduction.

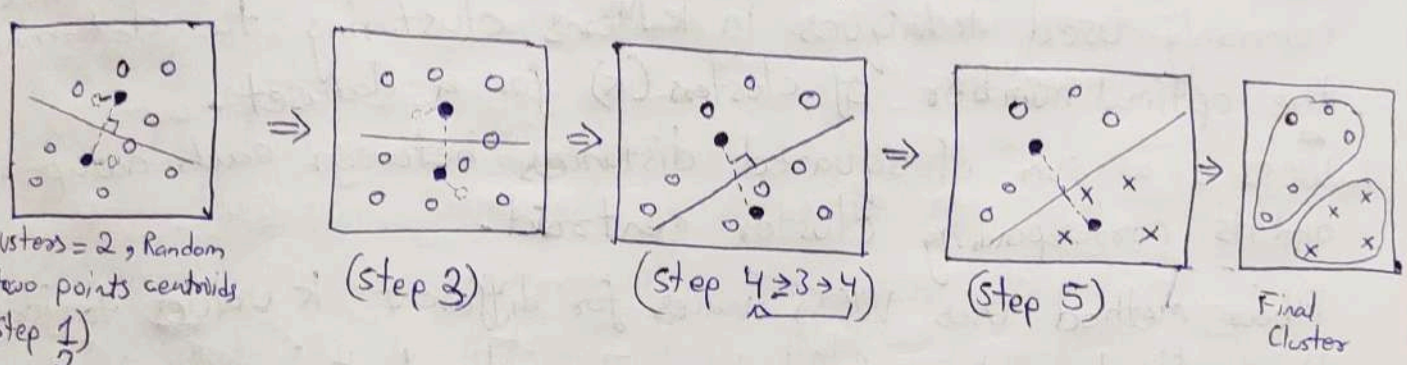
2025/4/27 10:59

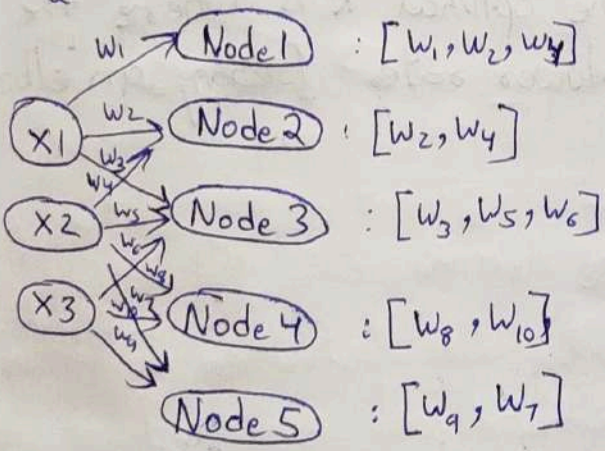
↳ K-Means clustering is an unsupervised ML algorithm used to partion a dataset into groups or clusters based on similarity. Each cluster contains more similar data points than those data points in other cluster.

K-Means Algorithm steps : 1. Choose number of clusters (k), 2. Initialize centroids, 3. Assign data points to the nearest centroid, 4. Update the centroids & repeat it, 5. Termination when there is no change in cluster assignment.

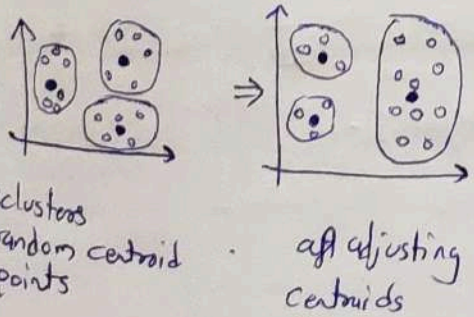
↳ SOMs retain topology of input set.
SOMs reveal correlations that are not easily identified.
SOMs classify data without supervision
No target vector → no backpropagation
No lateral connections between output nodes.

↳





↳ It is recommended to not using random Initialization trap.



To avoid this k-means ++ algorithm is used

2025/4/27 11:00

↳ k-Means++ is an improvement over the original kmeans algorithm designed to improve the initialization of centroids which helps avoid poor clustering results and speed up convergence.

Steps: Choose first centroid randomly, calculate distance b/w nearest selected centroid & each remaining point, choose next centroid based on probability distribution, repeat until k centroids are selected & finally run the standard kmeans algorithm from these initialized centroids.

↳ These kmeans & kmeans++ algorithms are not directly tied to RNNs, they may be used in preprocessing or feature extraction phase. Where clustering can help organize & simplify the data before its fed into RNN for sequential modeling.

↳ WCSS (Within cluster sum of squares) and Elbow method are commonly used techniques in k-Means clustering to determine the optimal number of clusters (k) for a dataset.

WCSS is a sum of squared distances between each datapoint and its corresponding cluster centroid.

Elbow method uses WCSS values for different k values to identify the optimal number of clusters. The optimal k is where the WCSS starts to decrease at a slower rate forming an 'elbow' in the plot.

12/02/25 Boltzmann Machines

Boltzmann Machines :

↳ It is a type of stochastic RNN & one of the earliest neural network models used in DL (Deep Learning). These are undirected probabilistic models that are used to learn the probability distribution of a set of input data, which makes them useful in tasks like data generation, feature learning & probabilistic inference.

↳ Concepts : EnergyBased models & Stochastic Nature

- EnergyBased models in Boltzmann Machines defines an energy function that assigns a scalar energy to each configuration of the network's state. The model tries to minimize this energy in order to find the most likely configuration of the network. The energy function for the network is designed in such a way that configurations with lower energy correspond to more probable states.
- Stochastic Nature : Neurons in a Boltzmann machines are stochastic means they are governed by probabilities rather than deterministic activations.

The probability is determined by Boltzmann distribution & is given by :

$$P(V_i = 1) = \frac{1}{1 + e^{-E(v)/kT}}$$

$V_i \rightarrow$ State of Neuron
$E(v) \rightarrow$ Energy of Configuration
$k \rightarrow$ Boltzmann constant
$T \rightarrow$ Temperature

- Boltzmann Machines consists of a set of neurons or units which are divided into visible units & hidden units.

Visible units represents input data. Hidden units are used to capture underlying structure or hidden factors that explain the input data. Visible & hidden units are connected to each other but there are no connections between units within same layer.

Connection blw the nerons are symmetric.

↳ Types : RBM (Restricted Boltzmann Machines)
DBM (Deep Boltzmann Machines)

RBM – These are simplified version of original Boltzmann machine in which connections b/w units within the same layers are restricted. No connections b/w visible units or between hidden units.

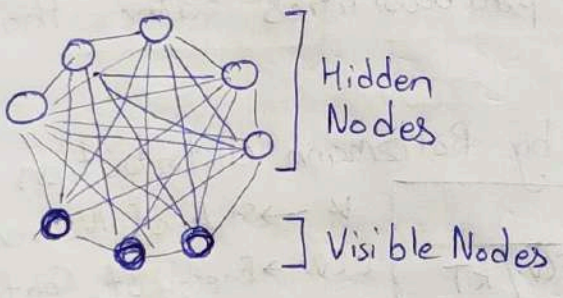
DBM – These are complex version of RBM, consisting of multiple layers of hidden units. Each layer is connected to the one below it allows network to learn a hierarchy of features from data.

↳ Boltzmann Energy function

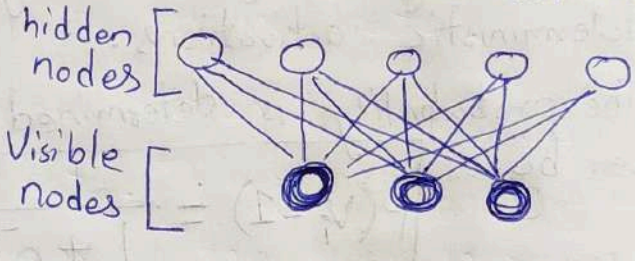
$$E(v,h) = -\sum_i b_i v_i - \sum_j c_j h_j - \sum_{i,j} W_{ij} v_i h_j$$

v_i – Vsisible units i ‖ b_i, c_j – Bias terms for visible & hidden units
h_i – Midden units j ‖ W_{ij} – The weight b/w visible unit i & hidden unit j.

↳ Boltzmann Machines



RBM (Restricted Boltzmann Machines)



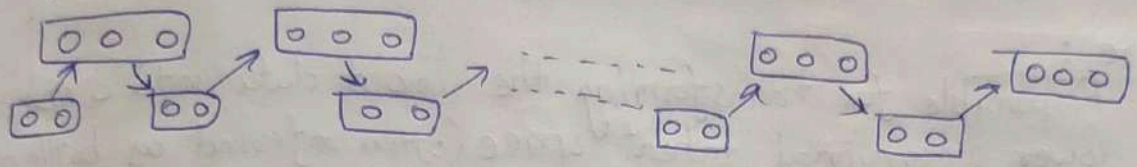
↳ Contrastive Divergence (CD) is a popular & efficient approximation algorithm used in training probabilistic models particularly for energy based models like Boltzmann Machines (BMs) and Restricted Boltzmann Machines (RBMs). It is to approximate the gradient of the log-likelihood of the data distribution, which is difficult to compare exactly due to the interactibility of calculating partion functions in energy based models.

The goal is to maximize the likelihood of the data.

2025/4/27 11:00

Contrastive Divergence algorithm approximates the true gradient of the log-likelihood by performing +ve phase (data phase), -ve phase (Model phase), weight update,
Applications : RBMs, Generative Models, Collaborative filtering, Image modelling

↳ Contrastive Divergence is a critical Algorithm in training energy-based models espicially RBMs. It enables efficient learning by approximating the complex gradient computation with Gibbs sampling. CD allows for the effective & scalable training of probabilistic models that would otherwise be too computationally expensive to train.



↳ Deep Belief Network is a type of generative deep Learning model that is composed of multiple layers of RBMs (Restricted Boltzmann Machines). It is designed to learn hierarchical features from the data by using unsupervised pretraining followed by supervised fine tuning. It is particularly useful in applications like image recognition, speech recognition & unsupervised ML.

↳ Deep Boltzmann (DBM) is a type of probabilistic generative model that is an extensive of the restricted boltzmann machines (RBM) to deep architecture. Similar to (DBNs), DBMs learn hierarchical representation of data by stacking multiple layers of Boltzmann machines. They are capable of generative new data and learning complex features. DBMs plays a significant role in development of generative models and unsupervised learning techniques.

13/02/2025 Auto Encoders

Auto Encoders

↳ It is a type of ANN(Artificial Neural Network) used to learn efficient representation of data, typically for the purpose of dimensionality reduction, feature extraction & unsupervised Learning.

↳ Autoencoder is to encode the input data into a compact, lower dimensional representation & then decode it back to the original data. The network is trained to minimize the difference between the input & the output, thus learning to capture the most feature of the input data.

↳ Architecture:

Encoder: Responsible for transforming the input data into a compact lower dimensional latent space (often referred as bottleneck or latent representation). This is achieved by applying one or more layers of neurons with activation functions like ReLu, Sigmoid, Tanh.

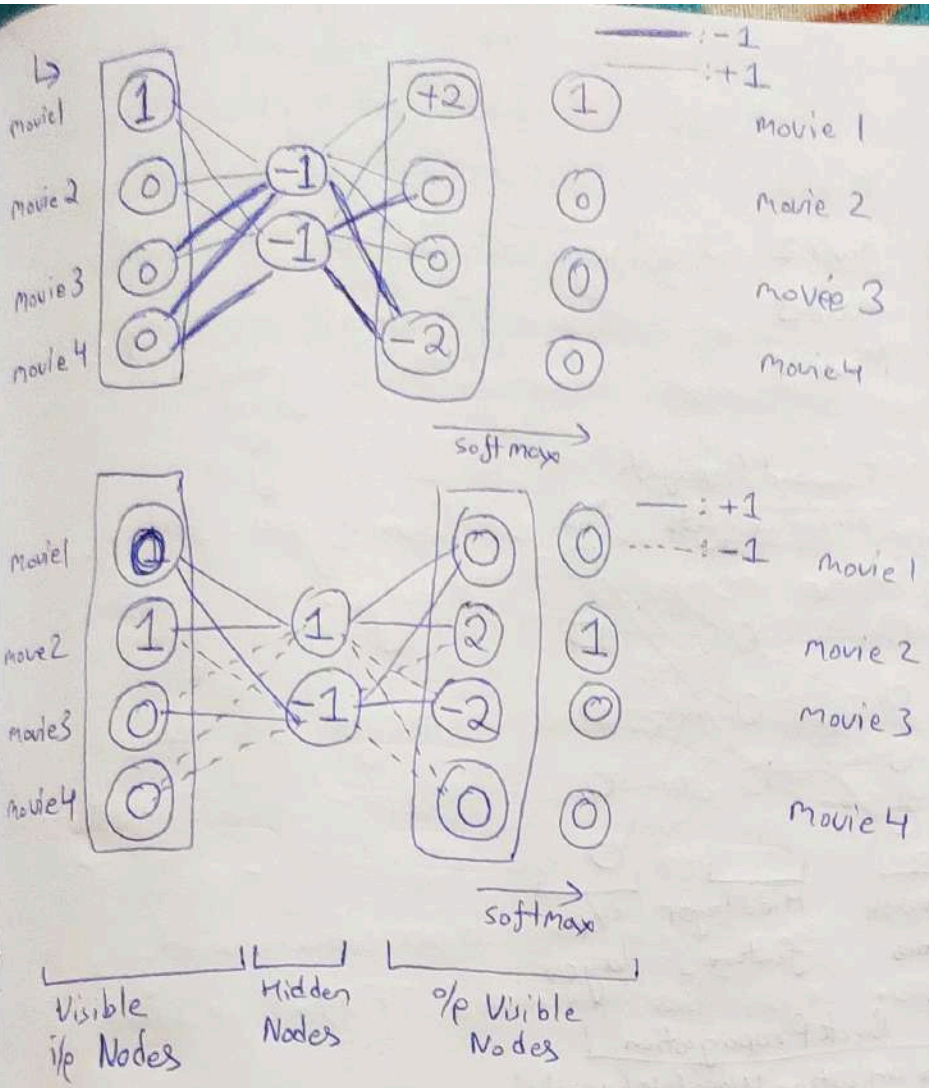
BottleNeck: Compressed representation of input data. It captures important features & eliminates redundancies, reducing the dimensionality of the original data.

Decoder: Takes the latent representation & reconstructs it back into the original input data. It is symmetric to encoder, with number of neurons in the layers increases as the data is reconstructed.

The goal is for output to be similar as possible to input.

↳ Application or Use cases: Dimensionality Reduction, Anamoly detection, Data Denoising, Image generation, Feature Learning, Image Compression & unsupervised Learning.

↳ Auto Encoders are powerful tool for Deep Learning Unsupervised Learning. Tanh activation function is commonly used in AutoEncoder ~~as the~~ in both the encoder & decoder parts of network.

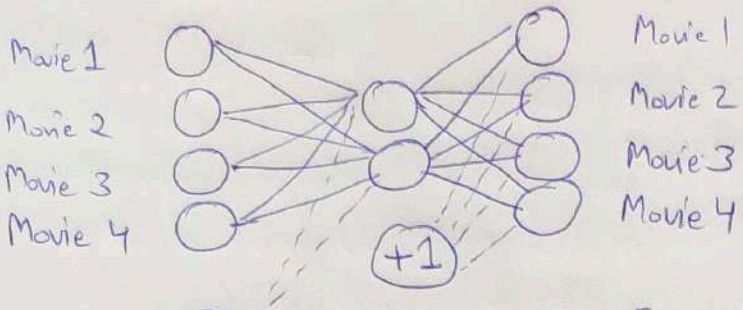


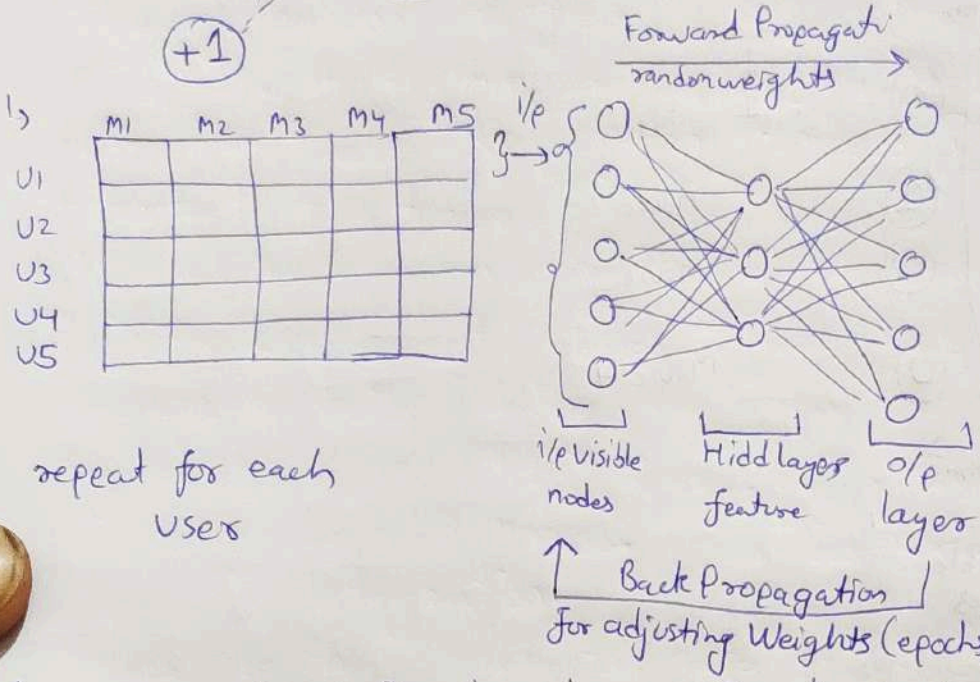
↳ Activation Functions are used to introduce non linearity into the Neural Networks. Some of them are Tanh, Sigmoid, ReLu.

- Tanh : $[-1, +1]$, zero centered symmetric, Stronger gradient than Sigmoid, Has vanishing Gradient, Computationally more expensive

Tanh	Sigmoid	ReLu
$[-1, +1]$	$[0, 1]$	$[0, \alpha]$
Zero centerd symmetric	Non Zero Centered Symmetric	Non Zero Centered Symmetric
Stronger Gradient than Symmetric	Gradient Vanishes	Very Stron gradient for +ve i/p , 0 for -ve i/ps
Has Vanishing Gradient	No Vanishing Gradient	No Vanishing Gradients
Computationally more expensive	Computationally more expensive	Very simple max function

↳ Biased AutoEncoder





repeat for each User

↳ Sparse Encoder in Deep Learning refers to a model or technique that creates a representation of data where most of the values are zeros or close to zero. Typically used where the data has lots of irrelevant or redundant features.

Denoising Encoder in DL is a model designed to remove noise from input data while keeping most important & useful information. Idea is to teach the model how to recover clean data from noisy data. It deals with noisy data by learning how to remove unwanted distortions and focus on essential features.

Contrastive encoder in DL is a model designed to learn a compact, efficient representation of input data while also making the learned features more robust to small changes or noise in the data.

Stacked Encoder in DL refers to a type of Neural Network architecture where multiple encoders are stacked on top of each other to learn hierarchical representation of data. It uses multiple layers of encoders to process data step by step, each layer learning more complex patterns from the previous one.

26/2/25

Tensorflow :-

↳ It is a low level software library created by Google to implement Machine Learning models and to solve complex numerical problems. It is a free & open source software library. It is symbolic math library based on data flow & differential programming.

↳ Tensors are multi dimensional arrays with uniform type. All tensors are immutable like python numbers & strings. We cannot update the content of a tensor.

↳ It is a powerful open soure library developed by Google for Deep Learning & Machine Learning. It provides flexible tools, libraries & community resources to build & deploy deep learning models efficiently.

Key features : Computational Graphs, Eager Execution, Scalability, Prebuilt models, Tensor Flow Extended (TFX), Keras API Integration.

Tensors are the fundamental data structures & used for computation. It is a multidimensional array (like Numpy Arrays) that allows Tensflow to perflow to perform efficient computations on CPUs and GPUs. Tensors are generalization of scalars, vectors and matrices.

0D - Single digit/numbers
1D - 1D array (Dimensional)
2D - 2D array (Dimensional)
Higher dimensional tensores i.e. 3D, 4D, ---,

API : Application Program Interface

Low Level API → It is generally more detailed and allows you to have more detailed control to manipulate functions within them on how to use and implement them.

High Level API → More generic and Simple and provides more functionality with one command statements than a low lower level API

2025/4/27 11:02

Keras

↳ It is high level deep learning (DL) API written in python for easy implementation and computation of neural networks. It is an open source software library that provides a tensorflow interface for Artificial Neural Networks. keras acts as an interface for the tensorflow library which means that it runs on the top of TensorFlow. Keras is high level api of tensorflow an approachable highly productive interface for solving Machine Learning problems with the focus on modern DL (Deep Learning).

↳ It is a high level DeepLearning API built into TensorFlow. It provides a simple interface for building, training & deploying DL models. Keras are user friendly, Modular, Scalable, Supports multiple Backends.

PyTorch

↳ PyTorch is a low level API developed by Facebook for NLP (Natural Language Processing) & Computer Vision. It is a powerful version of Numpy. It is an open source Machine Learning (ML) library based on the torch library. PyTorch is an open source DeepLearning (DL) frameworks developed by Facebook. It is widely used for research and production due to its dynamic computational graph and GPU acceleration. We use PyTorch for dynamic computational graph, GPU Acceleration, Autograd (Automatic Differentiation), Extensive libraries that Supports NLP, CV, RL etc...

↳ Highlevel interfaces are comparetively easier to learn and to implement models using them they allow you to write code in shorter amout of time and to be less involved with the details in this case tensorflow is a high and low level api. Pure tensorflow is low level api & Tensorflow wrapped in keras is a high level api.

2025/4/27 11:02

↳ Comparing Tensorflow, Keras PyTorch :

	Tensflow	Keras	PyTorch
Level of API	High & Low Level API	High Level API	Low Level API
Speed	Very fast, Used for high Performance	Slower than Tensorflow Tensor Flow as it works on to top of TensorFlow	Same speed as TensorFlow (Very Fast)
Architecture	Complex architecture Hard to Use	Simpler architecture as abstraction is used to make it simple to use.	Complex architecture
Datasets & debugging	Used for high performce models. De bugging is hard.	Used for smaller datasets debugging is easy & less frequent due to smaller models.	Used for large datasets. Easier to debug than TensorFlow.
Ease of Development	Hard to develope & write code	Easy to develope & is best for newbies.	Easier to learn than TensorFlow.
Ease of Deployment	Easy to deploy with TensorFlow serving	Model deployment can be done with TensorFlow serving or Flask	PyTorch mobile makes deployment easy but not as much as in TensorFlow.

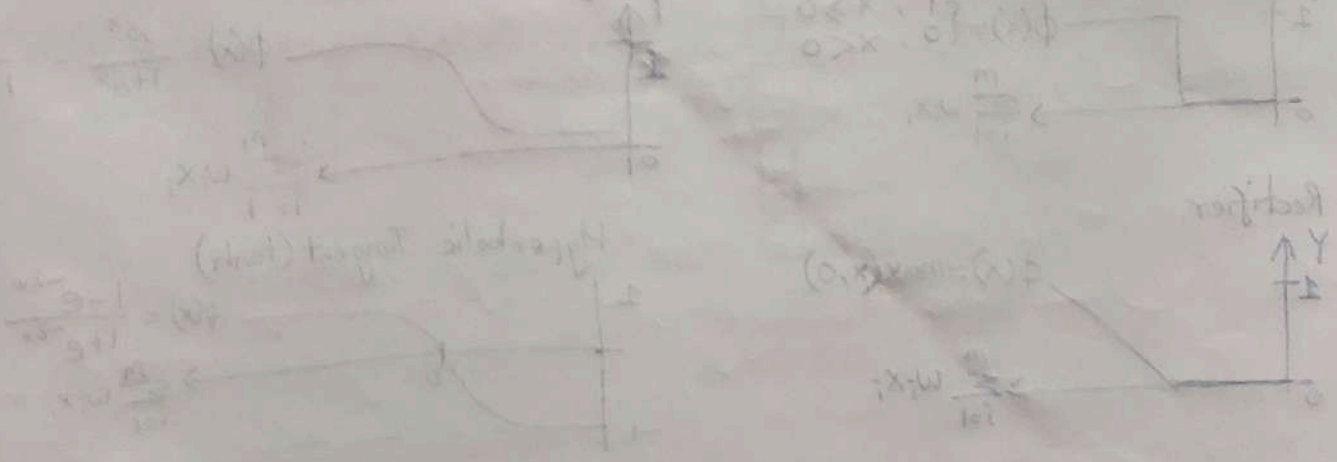
↳ TensorFlow has implemented various levels of abstraction to make implementation easy. This also makes debugging easy.

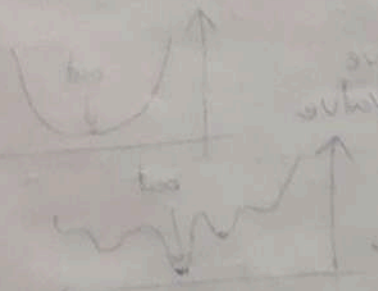
Keras is simple & easy but not as fast as TensorFlow. It is more user friendly (User) than any other Deep Learning API.

PyTorch is the preferred deeplearning API for teachers but is not as widely used in production as TensorFlow. It is Faster but lower GPU utilization.

2025/4/27 11:02

→ Deep Learning, Neural Network [i/p, hidden, o/p layers], neuron, Activation fun
→ ANN, CNN, RNN, SOM, BM, AE
→ FNN, LSTM, GRU, VAE, GANs, Transformer Network, Attension mechanism, ResNet, DenseNets, CapsNets, DBN, NTM, SNNs, Encoder-Decoder networks, U-Net.
→ Threshold, sigmoid





6/10/24

→ Geoffrey Hinton

→ Deep Learning : Neurons, neural networks, i/p layer, Hidden Layers, o/p layer.

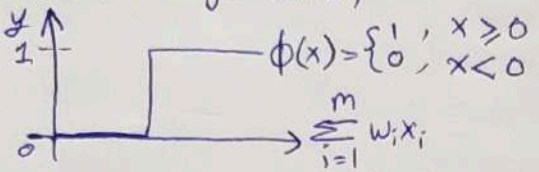
→ Supervised Learning : ANN - Artificial Neural Network for Regression & Classification
CNN - Convolutional Neural Network for Computer Vision
RNN - Recurrent Neural Network for Time Series analysis

UnSupervised Learning : SOM - Self Organising Maps for Feature detection
DBM - Deep Boltzmann Machines for Recommend Systems
AE - Auto Encoders for Recommend Systems.

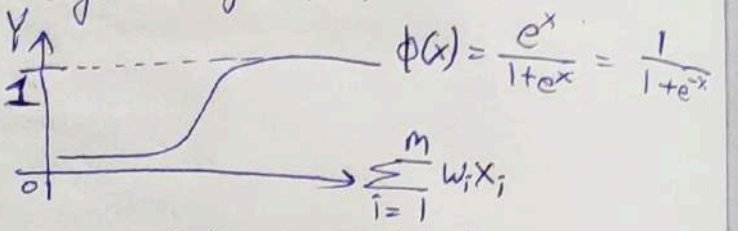
ANN

→ Activation Functions :

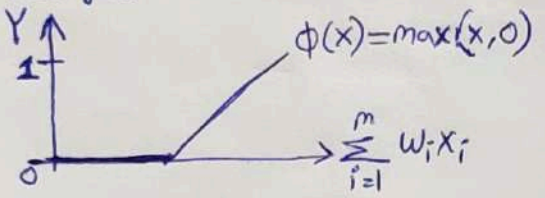
Threshold function



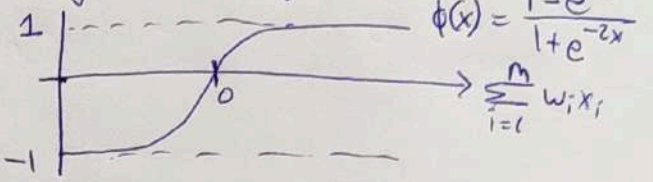
Sigmoid function



Rectifier



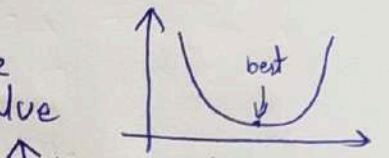
Hyperbolic Tangent (tanh)



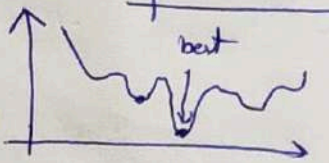
Step 1 : $\sum_{i=1}^{m} w_i x_i$, Step 2 : $\phi\left(\sum_{i=1}^{m} w_i x_i\right)$,

↳ How Neural Network learn ? → Throug training process includes forward & back propagation to adjust model's weights & minimize the error blw predicted & actual outcome.

↳ Gradien Descent : $C = (\bar{y} - y)^2/2$ || $\bar{y}$ → o/p value, y → actual value



stochastic Gradient descent : $C = \frac{1}{2}(\bar{y} - y)^2$



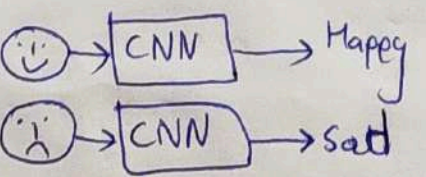
Forward Propagation : Neurons are activated from left to right.

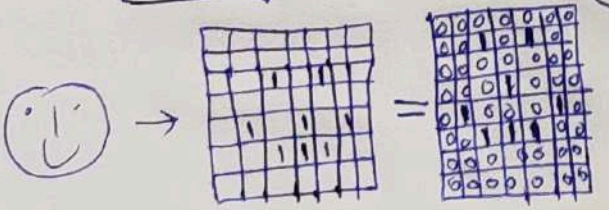
Back propagation : Error is propagated back from right to left.

CNN

↳ Convolution Neural Network

i/p → CNN → o/p say,





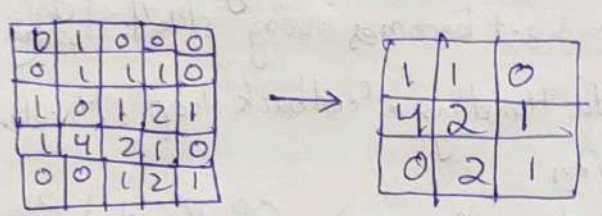
2025/4/27 10:36

↳ Steps : Convolution → Max Pooling → Flattening → Full Connection

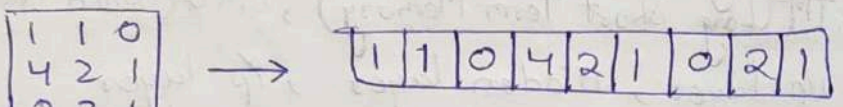
↳ Convolution : $(f*g)(t) \overset{def}{=} \int_{-\alpha}^{+\alpha} f(\tau)\, g(t-\tau)\cdot d\tau$

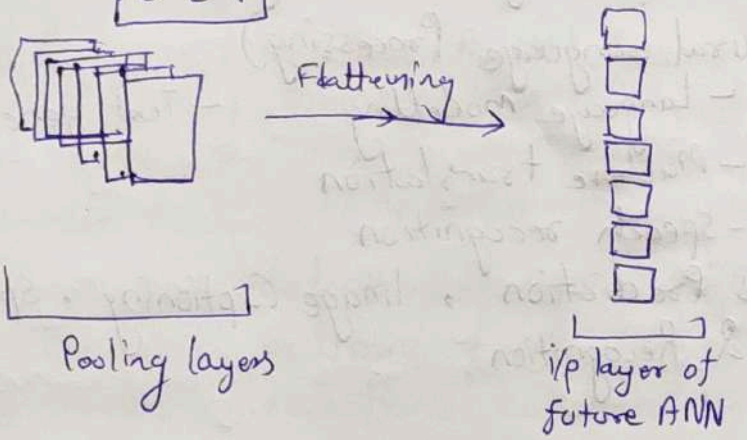
Relu layer : Rectifier $\phi(x) = \max(x, 0)$

Max Pooling :

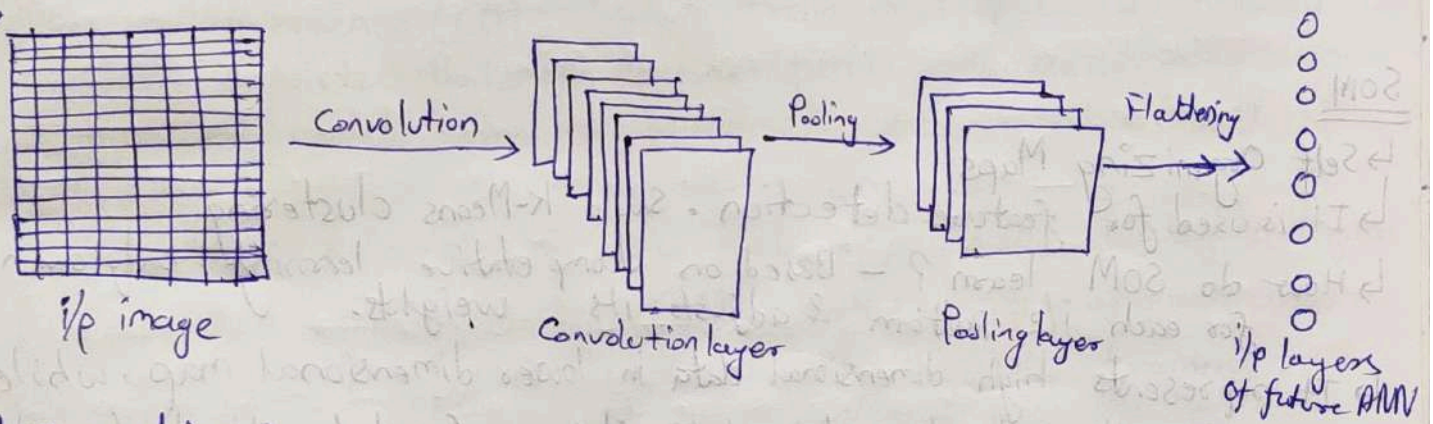


Flattening :

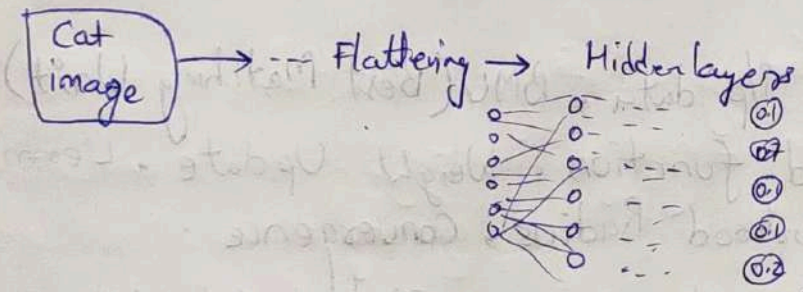




So,



Full Connection :



↳ Softmax & Cross Entropy : $f_i(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$, $L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$

$$H(p,q) = -\sum_x p(x) \log q(x)$$

2025/4/27 10:36

RNN

↳ Recurrent Neural Network : Type of ANN designed for processing sequences of data, such as time series or text. Desined to handle sequential data.

↳ Vanishing Gradient : Phenomenan that occours during training of deep Neural Network. It occours when gradient becomes very small during backpropagation.

↳ key concepts : Sequential Data Handling, Feedback loop, Hidden state, shared weights, vanishing gradien

Variants : LSTM (Long Short term Memory), GRU (Gated Recurrent Unit)

Components : i/p layer, Hidden layer, o/p layer

Applications : NLP (Natural Language Processing)
- Language modelling
- Machine translation
- Speech recognition
- Text generation

Time Series Prediction, Image Captioning, Speech Synthesis & Recognition

SOM

↳ Self Organizing Maps

↳ It is used for feature detection. say, K-Means clustering

↳ How do SOM learn? - Based on competitive learning, only one neuron for each i/p pattern & adjusts its weights.

↳ It represents high dimensional data in lower dimensional map, while preserving the topological relationships of data. Used for data clustering & visualization.

↳ concepts : Neuron grid, i/p data, BMU (Best Matching Unit), Neighbourhood function, Weight Update, Learning Rate, Neighbourhood Radius, Convergence.

Learning process : Initialization, i/p selection, BMU identification, weight update, Repeat.

characteristics : Topdogy Preservation, dimensionality reduction, clustering

Applications : Data Visualization, clustering, Anomaly detection, Pattern Recognition, Feature extraction

Elbow method : Graphical technique used in machine learning to determine the optimal number of clusters in a data set. Finding optimal value in a k-means clustering. Determine no. of centroids (k).

DBMs

↳ Deep Boltzmann Machines

↳ Energy based Models : $P_i = \frac{e^{-\varepsilon_i / kT}}{\sum_{j=1}^{M} e^{-\varepsilon_j / kT}}$

$$E(v,h) = -\left[\sum_i a_i v_i + \sum_j b_j h_j + \sum_i \sum_j v_i w_{ij} h_j\right]$$

$$P(v,h) = \frac{1}{Z} e^{-E(vh)}$$

↳ Restricted Boltzmann Machines (RBM): Type of neural network used for unsupervised learning, main feature extraction & dimensionality reduction tasks.

Deep Boltzmann Machines (DBMs): Advanced variant of RBMs, type of Deep learning model consists of multiple layers of hidden units. More complex patterns from data because they capture higher order interactions between variables.

Contrastive Divergence (CD): Used to train energy based models, espicially Restricted Boltzmann Machines (RBMs). To adjust the weights so that the model learns to approximate the distribution of i/p data.

Deep Belief Network (DBN): Generative mode composed of multiple layers of Restricted Boltzmann Machines (RBMs) stacked on top of each other.

Characteristics – Layered structures, Greedy layerwise training, Generative & discriminative, Feature learning

Advantages – Unsupervised PreTraining, Deep feature extraction, PreTraing helps in Optimization, Generative Power

Applications – Image & Video recognition, Speech recognition, Dimensionality reduction, Natural Language Processing

Auto Encoders Intuition

↳ Type of ANN used for unsperused learning tasks like dimensionality reduction, feature engineering/learning & data reconstruction.

2 main parts – Encoder, Latent Space (Bottleneck), Decoder

Key Concepts – Reconstruction, Bottleneck, Unsupervised Learning, Undercomplete Autoencoder

Applications – Dimensionality reduction, Denoising, Data compression, Feature Extraction, Anomaly detection, Generative models.

Types – Vanilla Autoencoder, Denoising Autoencoder (DAE), Sparse AutoEncoder, Variational AutoEncoder (VAE), Convolutional Autoencoder (CAE)

Limitation – Identifying Mapping, Data Specific, Lack of Interpretability.

↳ Sparse Autoencoders : Type of autoencoder that imposes a sparsity constraint on the hidden layer.

key Concepts – Sparsity, Activation Regularization, Overcomplete Representation

Applications – Feature extraction, Anomaly detection, Image processing, NLP, Pretraining

Advantages – Efficient Feature Learning, Reduced overfitting, Flexibility

DisAdvantages – Training Complexity, Sensitive to Hyperparameters.

↳ Denoising Autoencoder : Variant of autoencoder that is trained to remove noise from corrupted i/p data while preserving the original structure.

key concepts – corrupted input, Learning to remove noise, Improved Generalization

Training process – Corruption process → Loss function → Optimization

We use denoising encoder because of Robust feature learning, improved performance in real world applications, better generalization

Applications – Image Denoising, Anamoly detection, Pretraining, NLP (Natural Language Processing), Speech enhancement

Advantages – Nois robustness, feature learning, Generalization,

Disadvantages – Need for corrupted data, Training Complexity

↳ Stacked AutoEncoders : Type of Deep Learning model that consists of multiple layers of autoencoders stacked on top of another. It allows for learning increasingly complex & abstract features

2023/4/27 10:30

as the data passes through deeper layers. Helps to create a deep network that captures hierarchical representation of i/p data. making high effective for tasks such as feature learning, dimensionality reduction & pretraing for supervised tasks.

Key Concepts — Hierarchical Feature Learning, Deep architecture, UnSupervised pretraining

We use stacked Auto Encoders because of Capture Complex Features, Dimensionality reduction, Pretraining deep neural network, Improved Generalisation

Training process — Layerwise Pretraining → Unsupervised Pretraining → Supervised Fine tuning

Advantages — Hierarchical feature learning, Efficient Pretraining, Verasity, Better Initialization for Deep networks

Applications — Image recognition, Dimensionality reduction, Anomaly detection, Pretraining for Deep networks

↳ Deep Autoencoders: Advanced form of autoencoders where both the encoder & decoder components are composed of multilayers of neural networks. Allows model to capture more complex patterns and hierarchical representations in the data which makes it highly effective for tasks like dimensionality reduction, feature extraction & unsupervised pretraining for other deep Learning models.

Key Components — Encoder, Bottleneck, Decoder, Reconstruction Loss

We use Deep AutoEncoder because dimensionality reduction, feature extraction, Pretraining for supervised learning, Anomaly detection

Advantages — Non Linear dimensionality reduction, Hierarchical feature learning, Unsupervised learning, Improved performance for supervised tasks.

Applications — Dimensionality reduction, Anomaly detection, Pretraining deep networks, Denoising

2025/4/27 10:38

8/10/24

ANN (Artificial Neural Network) :-

↳ ANN contain artificial neurons which are called units. These units are arranged in a series of layers that together constitute the whole the ANN in a system.

↳ ANN has i/p layer, o/p layer as well as hidden layers.

↳ ANN is a ML model that mimics the structure & function of biological neural networks in the brain. It is made up of connected nodes or artificial neurons, that process data in a similar way to how neurons in the brain pass electrical impulses to each other.

↳ Applications – Pattern recognition, Forecasting, medicine, business, Pure sciences, data mining, Telecommunications & Operations management.

Key components of many types of AI (Artificial Intelligence), Large language Model (LLM) like chatgpt and AI image generator.

↳ Biological neuron : Dendrite, cell nucleus or Soma, Synapses, Axon.
Artificial neuron : inputs, nodes, weights, output

↳ ANN is a computational model inspired by the way biological neural networks in the human brain works. Key concept in ML and AI in DL.

↳ Key components : Neurons (Nodes), Layers, Weights, Activation functions, Bias

Works : Forward propagation, Loss function, Back propagation, Optimization algorithm.

Types of ANN : FNN (Feedforward Neural Network)
RNN (Recurrent Neural Network)
CNN (Convolution Neural Network)
DNN (Deep Neural Network)

Applications : Image recognition, NLP – [Natural Language Processing]
(Facial recognition) (Language translation, chatbots)
speech recognition, Autonomous driving,
predictive analysis (stock market prediction)

2025/4/27 10:38

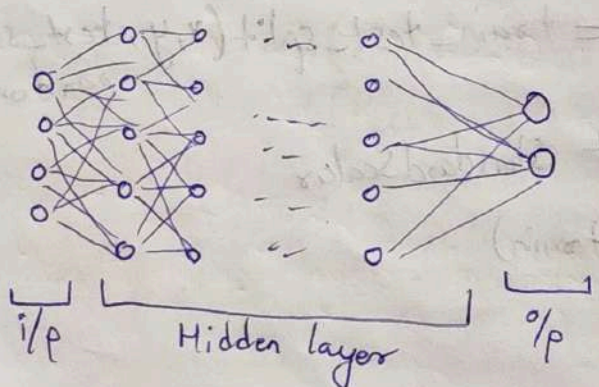
Code :-

```
import pandas as pd                                  # Importing packages
import numpy as np
import tensorflow as tf
print("Pandas version :", pd.__version__)
print("Numpy   version :", np.__version__)
print("Tensorflow version :", tf.__version__)

dataset = pd.read_csv("C:\\...\\...csv")             # Data preprocessing
x = dataset.iloc[:, 3:-1].values
y = dataset.iloc[:, -1].values
print(x)
print(y)
# Label Encoding
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
x[:, 2] = le.fit_transform(x[:, 2])
print("*"*20, "Label Encoding", "*"*20)
print(x)
print()
# OneHot encoding
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
ct = ColumnTransformer(transformers = [('encoder', OneHotEncoder(), [1])],
                       remider = 'passthrough')
x = np.array(ct.fit_transform(x))
print("*"*20, "One Hot Encoding", "*"*20)
print(x)
# Split data
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,
                                                    random_state=0)
# feature scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
x_train = sc.fit_transform(x_train)
x_test = sc.transform(x_test)
```

2025/4/27 10:38

```
# Adding the layers
ann = tf.keras.models.Sequential()                    # Neurons
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))  # 1st layer
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))  # 2nd layer
ann.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))  # outer layer
# Training dataset
ann.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])   # SDE
ann.fit(X_train, y_train, batch_size=32, epochs=100)
# Predictions
print(ann.predict(sc.transform([[1,0,0,600,1,40,.....,50000]])))
print(ann.predict(sc.transform([[1,0,0,600,1,40,.....,50000]]))>0.5)

y_pred = ann.predict(X_test)
print(np.concatenate((y_pred.reshape(len(y_pred),1),
                      y_test.reshape(len(y_test),1)),1))
y_pred = y_pred > 0.5
print(np.concatenate((y_pred.reshape(len(y_pred),1),
                      y_test.reshape(len(y_test),1)),1))
# Evaluation
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, y_pred)
print(cm)
print(accuracy_score(y_test, y_pred))
```

2025/4/27 10:39



binary_crossentropy → sigmoid
→ softmax

14/10/24

Convolution Neural Network (CNN) :-

↳ It is a type of ANN (Artificial Neural Network) that uses filters to learn and extract features from data such as images, text & audio. Primarily used for image recognition and processing, due to its ability to recognise patterns in images.

↳ It is a type of ANN designed primarily for processing structured grid data such as images. Tasks involving image recognition, video analysis, visual data due to their ability to automatically learn spatial hierarchies of features through layers.

↳ Key components : Convolution layer, ReLu (Rectified Linear Unit) Activation function, Pooling (Subsampling) layer, Fully connected (Dense) layer, Softmax (for classification).

Architechture : Input layer (i/p layer), convolution + Activation + Pooling Fully connected layer, % output layer

↳ Advantages : Spatial Hierarchies of features, parameter sharing, translation Invariance, Efficient Computation

Applications : Image classification, Object detection, Image Segmentation, Facial Recognition, video analysis

↳ Works : Input layers, Convolution layers, Activation function (Relu), Pooling (Down Sampling) layers, Multiple convolution + Pooling layers Flattening, Fully connected layers (Dense layers), % (output layer).

2025/4/27 10:39

↳ CNN are trained using back propagation & gradient descent like other Neural Networks.

Applications like: Image recognition, object detection, medical imaging, video processing, NLP (Natural Language Processing).

Code :

```
# Importing packages
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
print("Tensorflow version : ", tf.__version__)

# Data Preprocessing - Training set
train_data = ImageDataGenerator(rescale = 1./255, shear_range=0.2,
                                zoom_range = 0.2, horizontal_flip=True)
train_set_path= "C:\\----\\---"
train_set = train_data.flow_from_directory(train_set_path,
                                target_size =(64,64), batch_size =32,
                                class_mode = "binary")

# Data Preprocessing - Testing set
test_data = ImageDataGenerator (rescale = 1./255)
test_set_path = "C:\\---\\---"
test_set = test_data.flow_from_directory (test_set_path, target_size=
                                (64,64), batch_size =32, class_mode="binary")

# Building CNN - initializing CNN
cnn = tf.keras.models.Sequential()

# Convolution layers - 1st layer
cnn.add(tf.keras.layers.Conv2D(filters = 32, kernel_size = 3,
                                activation = 'relu', input_shape=[64,64]))
# Pooling - 1st layer
cnn.add(tf.keras.layers.MaxPool2D(pool_size = 2, strides = 2))

# Convolution layers - 2nd layer
cnn.add(tf.keras.layers.Conv2D(filters =32, kernel_size = 3,
                                activation = 'relu', input_shape = [64,64]))
# Pooling - 2nd layer
cnn.add(tf.keras.layers.MaxPool2D(pool_size =2, strides =2))
# Flattening
cnn.add(tf.keras.layers.Flatten())
```

- #Full connection
```
cnn.add(tf.keras.layers.Dense(units=128, activation='relu'))
```
- #Output layer
```
cnn.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
"""Note: Sigmoid for binary classification & Softmax for multiclass
    classification"""
```
- #Training CNN
```
cnn.compile(optimizer='adam', loss='binary_crossentropy',
            metrics=['accuracy'])
cnn.fit(x=train_set, validation_data=test_set, epochs=5)
```
- #Model prediction
```
import numpy as np
from keras.preprocessing import image
test_image = image.load_img("C:\\---\\--.jpg", target_size=(64,64))
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis=0)
result= cnn.predict(test_image)
train_set.class_indices
if result[0][0] == 1:
    prediction = "Dog"
else:
    prediction = "Cat"
print(prediction)
```
- #Display image
```
import matplotlib.pyplot as plt
img = plt.imread("C:\\---\\--.jpg")
plt.imshow(img)
plt.show()
```

2025/4/27 10:40

15/10/24

Recurrent Neural Networks (RNNs) :-

↳ It is a type of ANN (Artificial Neural Network) designed for processing sequential data, such as time series, language or data where order of information is necessary and audio.

↳ It has unique ability to maintain memory of previous inputs while processing new inputs.

↳ RNN is well suited for tasks like speech recognition, language modeling, time series forecasting.

↳ key concepts : Sequential data, Hidden states, Feedback loops

Types : Vanilla RNN, LSTM (Long Short Term Memory), Gated Recurrent Unit (GRU)

Use cases : NLP (Natural Language Processing), Time Series Analysis, Speech recognition, sequence classification

Key characteristics : Sequential processing, Hidden state, Feedback connections, Vanishing Gradient Problem

↳ LSTM : It is a type of RNN, designed to handle long term dependencies by solving the vanishing gradient problem.

LSTM architecture : cell state, Gates (Forget gate, input gate, output gate)

- Cell state :- Cell state is the memory of the network
- Gates :- Forget Gate deciceds which information from the cell should be discarded.

Input Gate determines what new information should be added to the cell state.

Output Gate controlls how much of the cell state is output as the hidden state at the current time step.

Advantages of LSTM : Handles Long Term dependencies & prevents vanishing Gradient problems

↳ Gated Recurrent Unit (GRU) : Type of RNN designed to address the vanishing gradient problem. It is simpler & computationally more efficient than LSTMs.

GRU architecture : Update Gate, Reset Gate,

Advantages : Simplicity, Performance

2025/4/27 10:40

↳ LSTMs are more complex with three gates (i/p, forget, o/p), which makes more flexible & potentially better for longer responses. GRUs are simpler with only two gates (Update & reset) leading to faster training. GRUs are often preffered when computational efficiency is a concern or when dealing with smaller datasets.

↳ Both LSTM & GRUs are widely used in applications like speech recognition, language translation & time series forecasting as they can capture the dynamics of sequential data effectively.

↳ If your data has very long dependencies & sufficient computational resources, LSTM might work better.

↳ If you need faster traing & a simpler model, GRUs may offer comparable performance with fewer computational costs.

Code :

```
# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing training dataset
dataset_train = pd.read_csv("C:\\...\\...csv")
training_set = dataset_train.iloc[:, 1:2].values
print(dataset_train.head())
print("*"*60)
print(training_set)

# Feature scaling - Standardisation & Normalization
from sklearn.preprocessing import MinMaxScalar
sc = MinMaxScalar(feature_range = (0,1))
training_set_scaled = sc.fit_transform(training_set)
print(training_set_scaled)
```

2025/4/27 10:41

```
# Creating a data structure with 60 timesteps and 1 output.
X_train = []
y_train = []
for i in range(60,1258):
    X_train.append(training_set_scaled[i-60:i,0])
    y_train.append(training_set_scaled[i,0])
X_train, y_train = np.array(X_train), np.array(y_train)
print(X_train)
print("*"*20)
print(y_train)

# Reshaping data
X_train = np.reshape(X_train,(X_train.shape[0],X_train.shape[1],1))
print(X_train)

# Building the RNN
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
regressor = Sequential()  # initialising RNN

# Adding 1st LSTM layer & some dropout regularisation
regressor.add(LSTM(units = 50, return_sequences = True,
                   input_shape = (X_train.shape[1],1)))
regressor.add(Dropout(0.2))

# Adding 2nd LSTM layer & some dropout regularisation
regressor.add(LSTM(units = 50, return_sequence = True))
regressor.add(Dropout(0.2))

# Adding 3rd LSTM layer and some dropout regularisation
regressor.add(LSTM(units = 50, return_sequences = True))
regressor.add(Dropout(0.2))
```

```
#Adding 4th LSTM layer and some dropout regularisation
regressor.add(LSTM(units=50))
regressor.add(Dropout(0.2))
# Adding output layer
regressor.add(Dense(units=1))
#Compiling RNN
regressor.compile(optimizer='adam', loss='mean_squared_error')
#Fitting RNN to training set
regressor.fit(x_train, y_train, epochs=100, batch_size=32)
# Getting real stock price
dataset_test = pd.read_csv("C:\\__\\__.csv")
real_stock_price = dataset_test.iloc[:,1:2].values
print(real_stock_price)
# Prediction - predicted stock price of 2017
dataset_total = pd.concat(dataset_train['Open'], dataset_test['Open'], axis=0)
inputs = dataset_total[len(dataset_total)-len(dataset_test)-60:].values
inputs = inputs.reshape(-1,1)
inputs = sc.transform(inputs)
X_test = []
for i in range(60,80):
    X_test.append(inputs[i-60:i,0])
X_test = np.array(X_test)
X_test = np.reshape(X_test,(X_test.shape[0], X_test.shape[1],
                            1))
print(X_test)
predicted_stock_price = regressor.predict(X_test)
predicted_stock_price = sc.inverse_transform(Predicted_stock_price)
print(Predicted_stock_price)
```

2025/4/27 10:41

```
# Visualising the results
plt.figure(figsize=[10,5])
plt.plot(real_stock_price, color='red', label='Real Google Stock Price')
plt.plot(predicted_stock_price, color='blue', label='Predicted
                                          Google stock Price')

plt.title("Google stock price prediction")
plt.xlabel("Time")
plt.ylabel("Google stock price")
plt.legend()
plt.show()
```

SOM (Self Organising Map) :-

18/10/2024

↳ It is a type of ANN (Artificial Neural Network), inspired by bidogical models of neural systems. It follows an unsupervised learning approach & trained its network through a competative learning algorithm.

↳ It is used for clustering & mapping techniques to map multidimensional data into lower dimensional which allows people to reduce complex problems for easy interpretation. It is has 2 layer input layer and output layer.

↳ It is also known as self organising feature Map (SOFM) is an unsupervised ML technique used to produce a low dimensional representation of a higher dimensional data set while preserving the topological structure of the data.

↳ Self Organising Map (SOM) is an unsupervised learning technique primarily used for dimensionality reduction & visualization. It organises high dimensional data into a 2D or 3D grid in such a way that similar data points are mapped close to each other.

↳ SOMs are useful in discovering patterns & relationships in complex data.

↳ key concepts : Input space, Output space, Weight vector, Competative learning, Neighbourhood function, Learning rate.

SOM Algorithm : Initialize, Select input vector, Best Matching Unit (BMU), Update weights, Repeat

Mathematical Steps : Compute the BMU ($d_i = \|x(t) - w_i(t)\|$)

Update the weights $\left[w_i(t+1) = w_i(t) + \alpha(t) \cdot h_{i,BMU}(t) \cdot (x(t) - w_i(t)) \right]$

Repeat the process

↳ Advantages : Dimensionality reduction, Visualization, Unsupervised Learning

Disadvantages : Computationally expensive, Paraneter Tuning, No objective function

↳ Kmeans & SOM are popular unsupervised learning algorithm used for clustering. Kmeans is a partition based clustering algorithm that aims to divide the dataset into a predefined no. of clusters (K). It does so by minimizing sum of squared distances blw the data points & their assigned clustor centre.

Code :

```
# Importing packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
dataset = pd.read_csv("c:\\__\\__.csv")
dataset.head()

# Feature Scaling
from sklearn.preprocessing import MinMaxScalar
sc= MinMaxScalar(feature_range(0,1))
x= dataset.iloc[:,:-1].values
y= dataset.iloc[:,-1].values
x= sc.fit_transform(x)

# Training SOM
from minisom import MiniSom
som = MiniSom(x=10, y=10, input_len= 15, sigma= 1.0, learning_rate =0.5)
som.random_weights_init(x)
som.train_random(data=x, num_iterations = 100)

# Visualizing results
from pylab import bone, pcolor, colorbar, plot, show
bone()
pcolor(som.distance_map().T)
colorbar()
markers=['o','s']
colors =['r','g']
for i,xv in enumerate(x):
    w= som.winner(xv)
    plot( w[0]+0.5 , w[1]+0.5,
          markers[int(y[i])],
          markedgecolor = colors[int(y[i])],
          markerfacecolor = 'None',
          markersize = 10,
          markedgewidth = 2
        )

show()
```

2025/4/27 10:42

```
#Finding Frauds
import numpy as np
mappings = som.win_map(X)
frauds = np.concatenate((mappings[(8,1)], mappings[(6,8)]), axis=0)
frauds = sc.inverse_transform(frauds)
print(frauds)
```

21/10/2024

Code : SOM + ANN

↳ Build a hybrid model, SOM (Self Organizing Map) & ANN (Artificial Neural Network)

```
#SOM (Self Organizing Map)
# importing packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
dataset = pd.read_csv("C:\\---\\---.csv")
dataset.head()

#Feature Scaling
from sklearn.preprocessing import MinMaxScalar
sc = MinMaxScalar(feature_range = (0,1))
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:,-1].values
X = sc.fit_transform(X)

# Training SOM
from minisom import MiniSom
som = MiniSom(x=10, y=10, input_len = 15, sigma=1.0, learning_rate=0.5)
som.random_weights_init(X)
som.train_random(data=X, num_iteration = 100)

#Visualizing results
from pylab import bone, pcolor, colorbar, plot, show
bone()
pcolor(som.distance_map().T)
colorbar()
markers = ['o','s']
colors = ['r','g']
```

```
for i,xv in enumerate(x):
    w= som.winner(xv)
    plot( w[0]+0.5 , w[1]+0.5 ,
          markers[int(y[i])],
          markeredgecolor = colors[int(y[i])],
          markerfacecolor = 'None',
          markersize =10,
          markeredgewidth =2
        )
show()
```

```
#Finding Frauds
import numpy as np
mappings = som.win_map(x)
frauds= np.concentrate((mappings[(8,1)], mappings[(6,9)]),axis=0)
frauds= sc.inverse_transform(frauds)
print(frauds)

#UnSupervised to Supervised DL model
#Creating matrix of features
customers = dataset.iloc[:,1:].values
#Creating dependent variable
is_fraud =np.zeros(len(dataset))
for i in range(len(dataset)):
    if dataset.iloc[i,0] in frauds:
        is_fraud[i] = 1
```

#ANN (Artificial Neural Network)

```
from sklearn.preprocessing import StandardScalar
sc = StandardScalar()
customerss = sc.fit_transform(customerss)

#Importing keras libraries & packages
from keras.models import Sequential
from keras.layers import Dense

#Initializing ANN
classifier = Sequential()
#Adding i/p input layer & 1st hidden layer
classifier.add(Dense(units=2, activation='relu'))
#Adding o/p layer
classifier.add(Dense(units=1, activation='sigmoid'))

#Traing the ANN (compiling ANN)
classifier.compile(optimizer='adam', loss='binary_crossentropy',
                   metricss=['accuracy'])
# Training the ANN (Training on training set)
classifier.fit(customerss, is_fraud, batch_size=1, epochs=2)

#Predicting the probabilities of fraud.
y_pred = classifier.predict(customerss)
y_pred = np.concentrate((dataset.iloc[:, 0:1], y_pred), axis=1)
y_pred = y_pred[y_pred[:, 1].argsort()]
print(y_pred)
```

22/10/24

Boltzmann Machine :-

↳ Boltzmann machines are a type of stochastic neural network particularly used for unsupervised learning tasks such as discovering patterns in data or pre training networks.

↳ Boltzmann Machine is a network of neurons(nodes) where each node is a binary unit (taking values 0 or 1), these are successfully connected to one another. Connection between nodes are undirected & symmetric which means any two nodes are bidirectional.

↳ It learns the probability distribution of the input data.

↳ key Features : Stochastic Binary units, Energy based model, Symmetric weights, Learning

Types : Standard Boltzmann machine, Restricted Boltzmann Machine (RBM)
Deep Belief Network (DBN)

Working : Initialization, Sampling, Energy minimization, Contrastive divergence

↳ Restricted Boltzmann Machines (RBM) : It consists of a visible layer (for the input data) & hidden layer (for learning features or patterns.

Steps to train : Foward Pass, Reconstruction, weight update

They are shallow, 2 layer neural nets that constitute the building blocks of deep belief networks. 1st layer called visible & 2nd, i/p layers called hidden layers.

Applications : Image recognition & NLP(Natural Language Processing)

↳ Deep Belief Network (DBN) : It is a type of DL algorithm that addressess the problems associated with classic neural networks. Used to solve unsupervised learning tasks to reduce the dimensionality of features and can be used to solve supervised learning tasks to build classification or regression models.

It is a type of ANN & it is known for ability to learn complex patterns & representations from large amount of data, particularly in unsupervised learning tasks.

2025/4/27 10:44

↳ Contrastive Divergence (CD) is a powerful algorithm that has facilitated the training of complex probabilistic models like RBMs.

↳ Application of Boltzmann Machines : Dimensionality reduction, Collaborative filtering, Feature learning, Generative models, Pre training in Deep Networks.

Advantages :- Good for unsupervised learning tasks (feature extraction, data generation)
- Can model complex dependencies b/w variables
- RBM are more efficient to train than traditional Boltzmann Machines.

Disadvantages :- Computationally expensive
traing will be slow, espicially with large networks.
DL techniques like deep neural network or Convolutional neural network (CNNs) have become popular for supervised learning tasks.

Code :

```
# Importing packages
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
from torch.autograd import Variable

#Importing dataset
movies = pd.read_csv("c:\\--\\--.csv", sep="::", header=None, engine='python',
                     encoding='latin-1')
users = pd.read_csv("c:\\--\\--.csv", sep='::', header=None, engine=
                    'python', encoding='latin-1')
ratings = pd.read_csv("c:\\--\\--.csv", sep='::', header=None, engine=
                      'python', encoding='latin-1')
```

2025/4/27

```
print(movies.head(), '\n')
print(users.head(), '\n')
print(ratings.head(), '\n')

# Preprocessing the training set and test set
training_set = pd.read_csv("c:\\...\\...csv", delimiter = '\t')
training_set = np.array(training_set, dtype='int')
test_set = pd.read_csv("c:\\...\\...csv", delimitter = '\t')
test_set = np.array(test_set, dtype='int')
print(training_set, '\n')
print(test_set, '\n')

# Getting number of users & movies
nb_User = int(max(max(training_set[:,0]), max(test_set[:,0])))
nb_movies = int(max(max(training_set[:,1]), max(test_set[:,1])))
print(nb_User)
print(nb_movies)

# Converting data into an array with users in lines & movies in colums
def convert(data):
    new_data = []
    for id_users in range(1, nb_users+1):
        id_movies = data[:,1][data[:,0]==id_users]
        id_rating = data[:,2][data[:,0]==id_users]
        ratings = np.zeros(nb_movies)
        ratings[id_movies-1] = id_ratings
        new_data.append(list(ratings))
    return new_data
training_set = convert(training_set)
test_set = convert(test_set)
print(training_set, '\n')
print(testing_set, '\n')
```

2025/4/27 10:45

```
# Converting the data into Torch tensors
training_set = torch.FloatTensor(training_set)
test_set = torch.FloatTensor(test_set)

# Converting the ratings into binary rating 1(liked) 0(not liked)
training_set[training_set == 0] = -1
training_set[training_set == 1] = 0
training_set[training_set == 2] = 0
training_set[training_set >= 3] = 1

test_set[test_set == 0] = -1
test_set[test_set == 1] = 0
test_set[test_set == 2] = 0
test_set[test_set >= 3] = 1

# Creating the architecture of Neural Network
class RBM():
    def __init__(self, nv, nh):
        self.W = torch.randn(nh, nv)
        self.a = torch.randn(1, nh)
        self.b = torch.randn(1, nv)
    def sample_h(self, x):
        wx = torch.mm(x, self.W.t())
        activation = wx + self.a.expand_as(wx)
        p_h_given_v = torch.bernoulli(p_h_given_v)
        return p_v_given_v, torch.bernoulli(p_h_given_v)
    def sample_v(self, y):
        wy = torch.mm(y, self.W)
        activation = wy + self.b.expand_as(wy)
        p_v_given_v = torch.sigmoid(activation)
        return p_v_given_v, torch.bernoulli(p_v_given_v)
```

2025/4/27 10:45

```
    def train(self, v0, vk, ph0, phk):
        self.W += torch.mm(v0.t(), ph0).t() - torch.mm(vk.t(), phk).t()
        self.b += torch.sum((v0 - vk), 0)
        self.a += torch.sum((ph0 - phk), 0)
nv = len(training_set[0])
nh = 100
batch_size = 100
rbm = RBM(nv, nh)
# Training RBM
nb_epoch = 10
for epoch in range(1, nb_epoch + 1):
    train_loss = 0
    s = 0.
    for id_user in range(0, nb_users - batch_size, batch_size):
        vk = training_set[id_user : id_user + batch_size]
        v0 = traing_set[id_user : id_user + batch_size]
        ph0, _ = rbm.sample_h(v0)
        for k in range(10):
            _, hk = rbm.sample_h(vk)
            _, vk = rbm.sample_v(hk)
            vk[v0<0] = v0[v0<0]
        phk, _ = rbm.sample_h(vk)
        rbm.train(v0, vk, ph0, phk)
        train_loss += torch.mean(torch.abs(v0[v0>=0] - vk[v0>=0]))
        s += 1.
    print('epoch: ' + str(epoch) + 'loss : ' + str(train_loss/s))
# Testing RBM
test_loss = 0
s = 0.
for id_user in range(nb_users):
    v = training_set[id_user : id_user + 1]
    vt = test_set[id_user : id_user + 1]
    if len(vt[vt>=0]) > 0:
        _, h = rbm.sample_h(v)
        _, v = rbm.sample_v(h)
        test_loss += torch.mean(torch.abs(vt[vt>=0] - v[vt>=0]))
        s += 1.
print('Test loss : ' + str(test_loss/s))
```

2025/4/27 10:46

25/10/2024 Auto Encoders

Auto Encoders :-

↳ It is a type of ANN(Artificial Neural Network) used in DL for unsupervised Learning.

↳ Its function is to learn a compressed representation of data, often for tasks like dimensionality reduction, denoising or anomaly detection.

↳ Structure : Encoder, Latent Space, Decoder

Encoder – First part of autoencoder that takes input data & compresses it into a lower dimensional representation

Latent Space – Compressed representation of data.

Decoder – Second part that takes the compresed representation & reconstructs the original input data.

↳ They are trained using a reconstruction loss, typically MSE (Mean Squared Error) for continous data or Binary cross entropy for binary data. Goal is to minimize the difference b/w i/p input & reconstructed output.

↳ Applications : Dimensionality reduction, Image Denoising, Anamoly detection, Generative models.

Auto encoders are a versatile tool in deep learning useful for both feature extraction & data generation.

↳ Sparse Encoders – Encourages sparcity in the hidden layer representation. Few neurons are activated at a time. Useful for feature extraction where only a subset of features is relevant, leading to more interpretable models. Captures important features.

Denoising Encoders – Trains the model to reconstruct clean inputs from corrupted versions. Effective for noise reduction in images or time series data, as it learn robust features that generalize well.

Contractive Autoencoder — It adds a penalty to the loss function that encourages the model to be robust to small changes in the input.
It is helpful in the scenarios where slight variation in input should not drastically change the representation like in image recognition.

Stacked Autoencoders — It consists of multiple layers of encoders & decoders stacked on top of each other.
It enhances feature learning capabilities & can capture complex hierarchical representation in the data.

Deep Autoencoders — It is a specific type of stacked autoencoder with many hidden layers, leading to a very deep network.
It is suitable for very complex data sets such as high resolution images or intricate sequential data.

↳ Autoencoders are a type of deep learning that are designed to recieve an input & transform it into a different representation.
Encoder converts i/p sequences into a 1D vector & decoder converts hidden vector into o/p sequences.

Code:

```
#Importing packages
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
from torch.autograd import Variable
#Importing dataset
movies = pd.read_csv("C:\\...\\...dat", sep="::", header=None, engine="python", encoding="latin-1")
Users = pd.read_csv("C:\\...\\...dat", sep="::", header=None, engine='python', encoding='latin-1')
ratings = pd.read_csv('C:\\...\\...dat', sep="::", header=None, engine='Python', encoding='latin-1')
```

2025/4/27 10:46

```
print(movies.head(), '\n')
print(users.head(), '\n')
print(rating.head(), '\n')

# Preparing training set & test set
training_set = pd.read_csv("c:\\---\\---.base", delimiter = '\t')
training_set = np.array(training_set, dtype = 'int')
test_set = pd.read_csv("c:\\---\\---.test", delimiter = '\t')
test_set = np.array(test_set, dtype = 'int')

# Getting number of users & movies
nb_users = max(max(training_set[:,0]), max(test_set[:,0]))
nb_movies = max(max(training_set[:,1]), max(test_set[:,1]))
print(nb_users)
print(nb_movies)

# Converting data into an array with users in lines & movies in colums
def convert(data):
    new_data = []
    for id_users in range(1, nb_users + 1):
        id_movies = data[:,1][data[:,0] == id_users]
        id_ratings = data[:,2][data[:,0] == id_users]
        ratings = np.zeros(nb_movies)
        ratings[id_movies - 1] = id_ratings
        new_data.append(list(ratings))
    return new_data
training_set = convert(training_set)
test_set = convert(test_set)

# Converting data into torch tensors
training_set = ~~convert~~ torch.FloatTensor(training_set)
test_set = torch.FloatTensor(test_set)
```

2025/4/27 10:47

```
# Creating the architecture of the Neural Network
class SAE(nn.Module):
    def __init__(self,):
        super(SAE, self).__init__()
        self.fc1 = nn.Linear(nb_movies, 20)
        self.fc2 = nn.Linear(20, 10)
        self.fc3 = nn.Linear(10, 20)
        self.fc4 = nn.Linear(20, nb_movies)
        self.activation = nn.Sigmoid()
    def forward(self, x):
        x = self.activation(self.fc1(x))
        x = self.activation(self.fc2(x))
        x = self.activation(self.fc3(x))
        x = self.fc4(x)
sae = SAE()
criterion = nn.MSELoss()
optimizer = optim.RMSprop(sae.parameters(), lr=0.01, weight_decay
                                                    =0.5)
# Training SAE
nb_epoch = 200
for epoch in range(1, nb_epoch+1):
    train_loss = 0
    s = 0.
    for id_users in range(nb_users):
        input = Variable(training_set[id_user]).unsqueeze(0)
        target = input.clone()
        if torch.sum(target.data > 0) > 0:
            output = sae(input)
            target.require_grad = False
            output[target == 0] = 0
            loss = criterion(output, target)
            mean_corrector = nb_movies/float(torch.sum(target.data > 0)
                                                + 1e-10)
            loss.backward()
            train_loss += np.sqrt(loss.data * mean_corrector)
            s += 1.
            optimizer.step()
print('epoch: ' + str(epoch) + 'loss: ' + str(train_loss/s))
```

```
# Testing SAE
test_loss = 0
s = 0.
for id_user in range(nb_users):
    input = Variable(training_set[id_user]).unsqueeze(0)
    target = Variable(test_set[id_user]).unsqueeze(0)
    if torch.sum(target.data > 0) > 0:
        output = sae(input)
        target.require_grad = False
        output[target == 0] = 0
        loss = criterion(output, target)
        mean_corrector = nb_movies/float(torch.sum(target.data
                                                   > 0) + 1e-10)
        test_loss += np.sqrt(loss.data * mean_corrector)
        s += 1.
print('test loss : '+ str(test_loss/s))
```

2025/4/27 10:47

28/10/24

Machine Learning

↳ ML algorithms are computational model that allows computers to understand patterns & forecast or make judgements based on data without explicit programming.

↳ Classification & Regression are supervised ML model techniques that use labelled data to predict outcomes.

In regression, the output is a continous or numerical value.

In classification, the output is a discrete or categorical value.

Model: MultiLinear

```
# Data preprocessing
# Importing libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Importing dataset
dataset = pd.read_csv("c:\\...\\....csv")
x = dataset.iloc[:,:-1].values
y = dataset.iloc[:,-1].values
print(dataset.head(), "\n")
print(x, "\n")
print(y, "\n")

# Data cleanup & handling missing data
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(missing_values= np.nan, strategy ="mean")
imputer.fit(x[:, 1:3])
x[:, 1:3] = imputer.transform(x[:, 1:3])
print(x)
```

```
#Encoding categorical data - Independent variable
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [0])],
                       remainder='passthrough')
X = np.array(ct.fit_transform(X))
print(X)

#Encoding categorical data - Dependent variable
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(y)
print(y)

#Split data
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                   random_state=1)
print(X_train, '\n')
print(X_test , '\n')
print(Y_train, '\n')
print(Y_test, '\n')

# Train data , Feature scaling
from sklearn.preprocessing import StandardScalar
sc = StandardScalar()
X_train[:, 3:] = sc.fit_transform(X_train[:, 3:])
X_test[:, 3:] = sc.fit_transform(X_test[:, 3:])
print(X_train, '\n')
print(X_test , '\n')
```

2025/4/27 10:48

Code 2 : Logistic

```
#Importing packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

#Importing dataset
dataset = pd.read_csv("C:\\...\\...csv")
print(dataset.head(), '\n')
x = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values
print(X, '\n')
print(y, '\n')

#Split data
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, y_test = train_test_split(X, y, test_size=0.25,
                                                    random_state=0)
print(X_train, '\n')
print(Y_train, '\n')
print(X_test, '\n')
print(Y_test, '\n')

#Feature scaling & training
from sklearn.preprocessing import StandardScalar
sc = StandardScalar()
x_train = sc.fit_transform(X_train)
X_test = sc.fit_transform(X_test)
print(x_train, '\n')
print(X_test, '\n')

from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(random_state=0)
classifier.fit(X_train, y_train)
```

```
#predicting model
print(classifier.predict(sc.transform([[30,87000]])))
#Predicting test result
y_pred = classifier.predict(x_test)
print(np.concaterate((y_pred.reshape(len(y_pred),1),
                      y_test.reshape(len(y_test),1)),1))

#Confusion Matrix
from sklearn.metrics import confusion_matrix, acuracy_score
cm = confusion_matrix(y_test, y_pred)
acc = accuracy_score(y_test,y_pred)
print(cm)
print(acc)
# Data Visualization
from matplotlib.colors import ListedColormap
x_set, y_set = sc.inverse_transform(X_train), y_train
x1,x2 =
```