NEURAL NETWORKS & DEEP LEARNING

★ Neural Network

- Machine (computational model) inspired by human brain, designed to perform tasks like pattern recognition, classification, prediction and decision-making by learning from data.

- Massively parallel system that learns from experience (training data), storing knowledge in weights (connections between neurons, the simple processing units)

— Properties

① Nonlinearity (can model complex relationships)
② Adaptive to new data
③ Can be implemented in hardware (VLSI)
④ Works even if some neurons fail (fault tolerance)

★ Human Brain

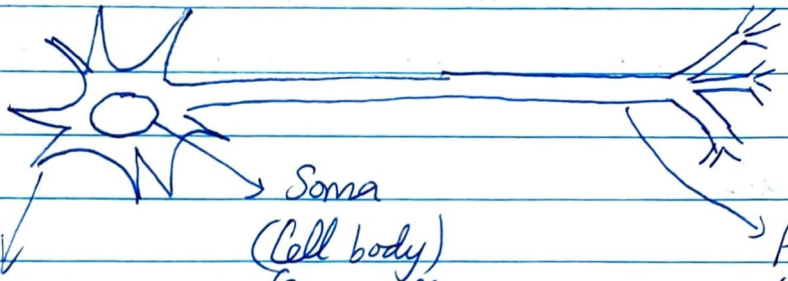
- The human nervous system consists of 3 stages that processes information and produces responses:

① Receptors (Recieve stimuli/input from the environment)
② Neural Network (brain) (processes information and makes decisions)
③ Effectors (produces responses) (muscles, glands, actions)

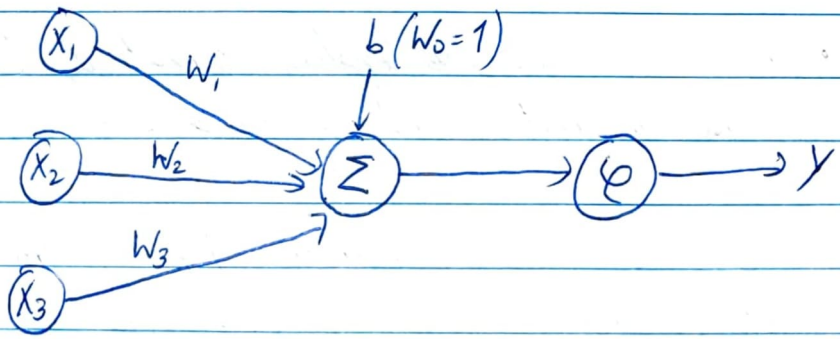
- Information flows forwards ⟶
  Feedback exists to improve performance.

- Neurons are the basic structural and functional units of the brain responsible for recieving, processing and transmitting information.

- Learning in the brain happens by changing the strength of connections between neurons. (synaptic weights)



- When summed inputs exceeds a threshold, the neuron fires (Artificial Neural Network (ANN))



- Synaptic Weights: Each input signal x_i is associated with a weight w_i representing the strength of a connection between neurons.

- **Summation Function:** Neuron computes weighted sum of all inputs and bias is added to improve flexibility (Induced local field)

$$v = \sum w_i x_i + b$$

- **Activation Function:** Determines output of neuron by introducing non-linearity into the network

① **Threshold / Heaviside function**

- Output is binary $\{0, 1\}$. $y = \varphi(v) = \begin{cases} 1 & \text{if } v \geq 0 \\ 0 & \text{if } v < 0 \end{cases}$
- Not differentiable, hence cannot be used with backpropagation

② **Sigmoid function**

$$y = \varphi(v) = \frac{1}{1 + e^{-av}}$$

$a \rightarrow$ controls slope

- Output between $[0, 1]$
- Smooth and differentiable
- Not zero-centered, ~~zero~~ and suffers from vanishing gradient

③ **Tanh** (Hyperbolic Tangent)

$$y = \varphi(v) = \frac{e^{v} - e^{-v}}{e^{v} + e^{-v}}$$

- Output between $[-1, 1]$
- Zero-centered
- But still suffers from vanishing gradient.

Note: Not being zero-centered is an issue as it restricts the

optimizers ability to make balanced weight updates, making training less efficient. Causes skewed updates and zig-zag convergence towards optimum.

④ ReLU (Rectified Linear Unit)

$$y = \varphi(v) = \max(0, v)$$

- Output between $[0, \infty)$
- Reduces vanishing gradient problem
- Very fast but can lead to 'dead' neurons.

⑤ Softmax

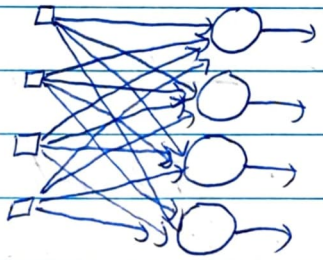
$$y = \varphi_i(v) = \frac{e^{v_i}}{\sum e^{v_j}}$$

- Used for multi-label classification, in output layer only
- Converts outputs into probabilities, all of which sum to 1.

☆ Network Architecture

- Describes how neurons are arranged and connected in a neural network and how signals flow between them.

① Single-Layer Feedforward Network (Rosenblatt's perceptron)

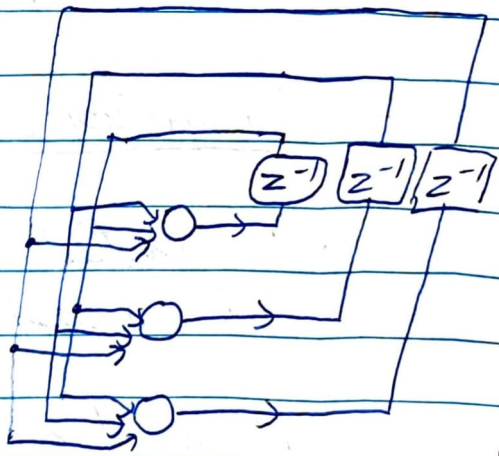


- Consists of input layer (source nodes) and output layer (computation nodes)
- No hidden layers.
- Signals flow in only one direction
- No feedback connections.
- Can classify only linearly separable patterns, not XOR.

② Multi-layer Feed-Forward Network (MLP)

- Consists of input layer, one or many hidden layers and an output layer
- Signal flows layer by-layer in forward direction.
- Hidden neurons extract higher-order features and perform non-linear transformation, thus enabling the network to solve complex, non-linearly seperable problems.

③ Recurrent Neural Network



- Contain feedback loops
- Output of a neuron may be fed back as input to other neurons
- No self-feedback
- z^{-1} → unit time delay
- Network behaviour depends on current input and past inputs (memory)
- Applications include speech recognition, NLP, time-series prediction.

☆ Learning

- Process by which a neural network adjusts its synaptic weights based on experience, so that performance improves over time.

— Supervised Learning (with a teacher)

- Teacher provides the correct (labelled) output/data/knowledge

for each input and once trained, the network can operate without a teacher.

- Weights are updated based on error between desired and actual outputs (iteratively)

[Error = Desired Output − Actual Output]

Ex- Handwritten digit recognition, pattern recognition.
Models include perceptron, FFNN, MLP

– Unsupervised Learning (without teacher)

- No labelled data is present and learning based on task-independent quality measure.
- Network discovers patterns and structures in the input data
- Mainly used for clustering and feature discovery.

– Reinforcement Learning

- No explicit desired output is provided
- Network learns optimal actions through reward-based feedback while interacting with the environment

★ Learning Tasks

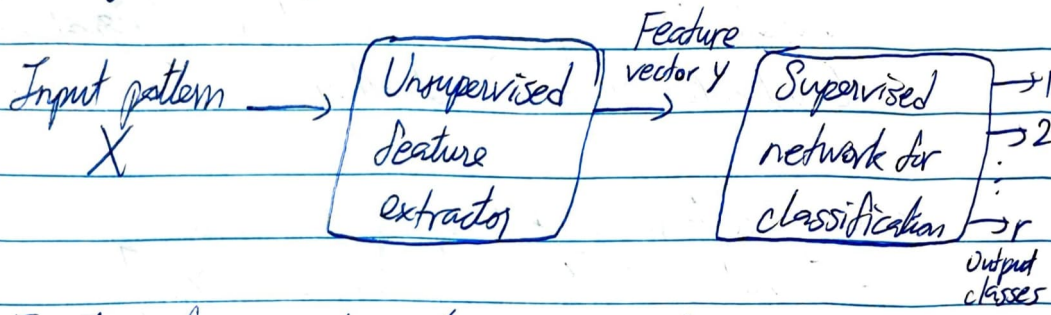
- What the neural network is expected to learn or perform.

- Pattern Association

- Learning by associating patterns
- Autoassociation (unsupervised learning) (I/O patterns same)
- Heteroassociation (supervised learning) ((I/O) pattern diff)

- Pattern Recognition

- Assign of input pattern to a class
- Network is trained with labelled data, enabling it to classify unseen patterns



- Function Approximation (IO mapping)

- Uses supervised learning to learn unknown non-linear function
- Used in system identification and inverse modelling.

- Control

- Used to regulate plants or systems, that can handle non-linearity, noise and parallel control actions.

- Beamforming

~~for each input and once trained, the network can operate without a teacher.~~

- ~~Weights are updated based on error between~~
- Used in radar and sonar systems to separate target signal from noise.
- These networks adaptively optimize beam patterns.

Rosenblatt's Perceptron (1958)

- Simplest form of a neural network designed for pattern classification (binary) if they are linearly separable
- A perceptron consists of:
  - Input vector $[x_1, x_2, x_3, \ldots, x_m]$
  - Synaptic weights $[w_1, w_2, w_3, \ldots, w_m]$
  - Bias b ~~(weight input)~~ (input = 1)
  - Activation function (threshold)
- Perceptron creates a decision surface that separates the two classes

$$v = w^T x + b, \quad y = \varphi(v) = \begin{cases} +1 & \text{if } v \geq 0 \\ -1 & \text{if } v < 0 \end{cases}$$

$+1 \rightarrow$ assign to class C_1,
$-1 \rightarrow$ assign to class C_2

- Hyperplane, $(w^T x + b = 0)$ 2D → straight line, 3D → plane

- Weights updated when perceptron misclassifies an input, decision boundary is shifted until all training samples are classified correctly.

(Pros & Features)
- Uses single neuron, simple and computationally efficient
- Binary classification only, foundation for MLP

(Cons)
- Classifies only linearly separable data, fails for XOR
- Uses non-differentiable activation function, cannot be trained using backpropagation.

★ Perceptron Convergence Theory

"
- If two classes are linearly separable, then perceptron learning will converge in a finite number of steps.
- Convergence is not guaranteed if the hyperplane cannot separate two classes completely.
- Bias b is treated as a synaptic weight.

$$x = [1, x_1, x_2, x_3, \ldots, x_m]^T$$
$$w = [b, w_1, w_2, w_3, \ldots, w_m]^T$$

- Hyperplane : $(w^T x = 0)$
- During the learning process, training samples are presented one-by-one.
  If sample is correctly classified, weights remain unchanged
  If sample is misclassified, weights are updated.

(predicted)
If $y(n)$ → actual output, $d(n)$ → desired output.
η → learning-rate parameter (determines how fast neuron should learn)

$$\begin{aligned} w_{new} &= w_{old} + \eta \times [d(n) - y(n)] \times x_n \\ b_{new} &= b_{old} + \eta \times [d(n) - y(n)] \end{aligned}$$

☆ Wiener Filter

- Linear filter that provides the optimal weight vector by minimizing the MSE between desired and filter output

$$y(n) = w^T x(n), \quad e(n) = d(n) - y(n)$$

$$MSE = E[e^2(n)] = E[(d(n) - w^T x(n))^2]$$

where E → expectation operator (average across all possible noisy inputs)

- The optimal Wiener weight vector, upon minimizing MSE leads to:

$$\begin{aligned} w_o &= R^{-1} p \\ &= [X(n) X^T(n)]^{-1} [X(n) d(n)] \end{aligned}$$

where $E[x(n) x^T(n)]$ → autocorrelation matrix of input
$E[x(n) d(n)]$ → cross-correlation vector.

- Requires complete statistical knowledge of input signal and desired signal. (High complexity, less practical)
- Not practical for adaptive environments

Ex: Compute Wiener filter for AND gate.

x_1	x_2	d
0	0	0
0	1	0
1	0	0
1	1	1

$\left(y = w^T x = w_1 x_1 + w_2 x_2\right)$

$R = E[x(n)x^T(n)]$,

$[0\ 0]^T[0\ 0] = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$ / $[0\ 1]^T[0\ 1] = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$

$[1\ 0]^T[1\ 0] = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$ / $[1\ 1]^T[1\ 1] = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$

$R = \Sigma x x^T = \frac{1}{4} \times \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 0.5 & 0.25 \\ 0.25 & 0.5 \end{bmatrix}$

$p = E[x(n)\,d(n)]$

$[0\ 0].0 = [0\ 0]$, $[1\ 0].0 = [0\ 0]$

$[0\ 1].0 = [0\ 0]$ $[1\ 1].1 = [1\ 1]$

$p = \frac{1}{4} \times \Sigma x d = \frac{1}{4} \times [1\ 1] = [0.25\ 0.25]$

$|R| = (0.5)(0.5) - (0.25)(0.25) = 0.1875$

$R^{-1} = \frac{1}{0.1875} \begin{bmatrix} 0.5 & -0.25 \\ -0.25 & 0.5 \end{bmatrix}$

$$W_o = R^{-1}p = \frac{1}{0.1875}\begin{bmatrix} 0.5 & -0.25 \\ -0.25 & 0.5 \end{bmatrix}\begin{bmatrix} 0.25 \\ 0.25 \end{bmatrix} = \begin{bmatrix} 0.33 \\ 0.33 \end{bmatrix}$$

$\therefore W_1 = W_2 = 0.33$

Verify: For $(1,1)$, $y = 0.33(1) + 0.33(1) = 0.66 \approx 1$

Least Mean Square Algorithm

- Adaptive algorithm that iteratively estimates the Weiner solution using instantaneous data samples. (Stochastic Gradient Descent method)
- Despite having same objective as Wiener filter, here instead of computing expectations, it uses sample-by-sample updates.

$$\hat{w}(n+1) = \hat{w}(n) + \eta\, x(n)\, e(n)$$

for each iteration n

- Learning rate (η) controls speed and stability. (It's inverse acts as measure of memory)
- If η is small, slow convergence though accurate learning.
- If η is large, leads to faster convergence and risk of instability and divergence.

Note: While Wiener strives for exact optimal solution, LMS adaptively approximates solution around the Wiener.

☆ Markov Model

- Weight-Error (Deviation) vector: $\Sigma(n) = w_o - \hat{w}(n)$

where $w_o \rightarrow$ optimal Wiener weight vector
$\hat{w}_n \rightarrow$ LMS weight vector at iteration n

- If $\Sigma(n) = 0 \rightarrow$ (LMS has converged)
  If $\Sigma(n) \neq 0 \rightarrow$ LMS is still learning
  This vector measures how far LMS is from Wiener solution

- The equation for Markov model of LMS deviation is:

$$\Sigma(n+1) = A(n)\,\Sigma(n) + f(n)$$

- Transition Matrix, $A(n) = I - \eta\, x(n)\, x^T(n)$

Determines how much of current error remains, thus controlling convergence / divergence ($I \rightarrow$ identity matrix)

- Noise / Driving Force Term, $f(n) = -\eta\, x(n)\, e_o(n)$

where $e_o(n) \rightarrow$ Wiener estimation error.

$$e_o(n) = d(n) - w_o^T x(n)$$

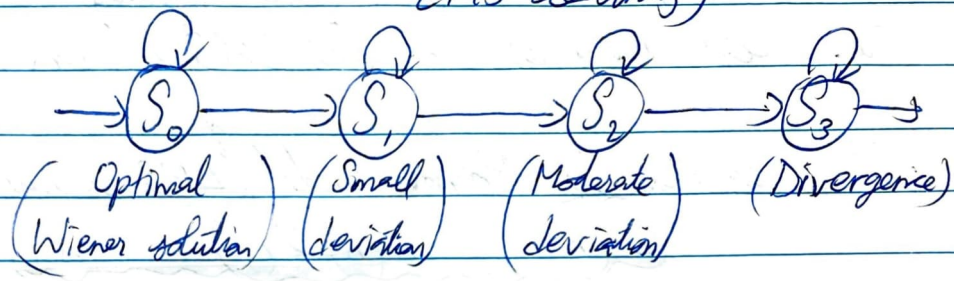
Exists due to noise in the environment, which cannot be eliminated completely, causing LMS to fluctuate even after convergence.

- A process is Markov if:

$$P(\varepsilon(n+1) \mid \varepsilon(n), \varepsilon(n-1), \ldots) = P(\varepsilon(n+1) \mid \varepsilon(n))$$

which means LMS deviation at time $(n+1)$ depends only on deviation at time (n), past is irrelevant.

- $[z^{-1}\,\varepsilon(n+1) = \varepsilon(n)]$ where z^{-1} → unit delay element (Represents memory & feedback in LMS learning)



★ Multi-Layer Perceptron

- Rosenblatt's perceptron → single-layer network
  LMS algorithm → single linear neuron.
  Both can only solve linearly-separable problems

- MLP is a feedforward neural network that contains an input layer, one/multiple hidden layers and an output layer.

- Each neuron includes a differentiable non-linear activation (sigmoid) as the NN is trained using backpropagation (gradient descent)

- Neurons are interconnected through synaptic weights, which determine learning capability

- Hidden neurons act as feature detectors that transform input data into a new space called feature space

- Input Layer: Made of sensory units that supply input vectors to the network. No computation happens here.

- Hidden Layers: Perform intermediate computations.
  Output Layer: Produces final response of the network

- Forward-pass Signals originate at the input layer, propagate forward through the network, are computed as functions of input signals & synaptic weights and finally appear at the output layer as network output.

- Backward/Error Signals originate at the output layer, are computed using desired & actual output, propagate backward layer-by-layer to adjust weights during learning.

☆ • Learning Modes

- How and when synaptic weights are updated during training of a neural network.

— Batch Learning

- Synaptic weights are updated after all training samples are presented.

- Learning is performed over an entire epoch.
- Cost function, $\left(\mathcal{E} = \frac{1}{N} \sum e^2(n)\right)$ (Average error)

(Pros) • Uses accurate gradient, smooth learning curve
(Cons) • However requires high storage, less adaptive to change and computationally expensive

– <u>On-line Learning</u> (Stochastic)

- Synaptic weights are updated after each training sample.

$$\left(\mathcal{E}(n) = e^2(n)\right)$$ (instantaneous error)

- An epoch consists of presenting all N samples once. Samples are often randomly shuffled after each epoch.

(Pros) • Requires low storage, highly adaptive, can track non-stationary environment.
(Cons) • Noisy learning curve, noisy (stochastic) gradient.

- Less likely to get trapped in local minima and better suited for real time learning.

Note: To improve speed, stability and accuracy of learning

1. Proper initialization of weights
2. Proper choice of learning rate η
3. Momentum adds fraction of previous weight update to current update; to smoothen learning and reduce oscillations

④ Data Normalization → [0, 1] (or) [-1, 1]
⑤ Using suitable activation function
⑥ Avoiding overfitting
(Training too long can cause model to memorize data)
(Early stopping, proper network size, adequate data)

Note: In MLP, to compute error for:

Output layer neuron: $\delta_k \leftarrow O_k(1-O_k)(t_k - O_k)$

Hidden layer: $\delta_h = O_h(1-O_h)\sum \delta_k W_{hk}$

$W_{new} = W_{old} + \eta \cdot \delta \cdot input$ (Weight updation)

where O_k → output of output neuron
O_h → output of hidden neuron.
W_{hk} → weights (hidden ↔ output)
W_{ih} → weights (hidden ↔ inputs)
t_k → target / desired output

MODULE - 2

- To classify patterns that are not linearly seperable, instead of using backpropagation like MLP, we can also follow this hybrid approach:

(i) Nonlinear Transformation (Unsupervised Learning)

Transform input data into higher dimensional space using hidden non-linear functions, thus making non-linear data linearly seperable.

(ii) Linear Classification (Supervised learning)

Once Transformed, use Least Squares estimation to train output layer (linear).

- Cover's Theorem: A complex classification problem is more likely to be linearly seperable in high dimensional space than in low-dimensional space (if space isn't densely populated)

- Feature space = high dimensional mapped space
  Hidden functions (ϕ) = Non-linear function in hidden layer.
  Linear Seperability = Classes seperated by hyperplane

- This approach is preferred over MLP as training is faster, no backpropagation needed making it computationally efficient. Output weights can be solved directly.

- Suppose input space dimension = m_0 and after mapping, new dimension = m_1, such that $m_1 > m_0$.
When dimension increases, the probability of finding the seperating hyperplane increases, making the data easier to classify.

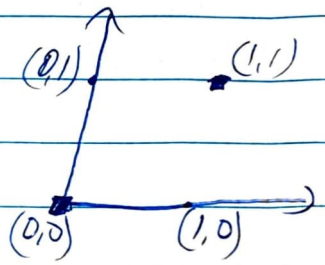
- Data is linearly seperable if:
$$w^T \phi(x) > 0 \quad \text{for class } H_1$$
$$w^T \phi(x) < 0 \quad \text{for class } H_2$$
The seperating surface : $w^T \phi(x) = 0$
where $x \in R^{m_0}$
$$\phi(x) = [\phi_1(x), \phi_2(x), \ldots, \phi_{m_1}(x)]^T \text{ (mapping func)}$$
This maps : $R^{m_0} \to R^{m_1}$ ↳ hidden non-linear functions

- Linear classifier/hyperplane in higher dimension = Nonlinear classifier/hypersurface in original space.

★ XOR problem

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

(different = 1, same = 0)



(not linearly seperable in 2D space)

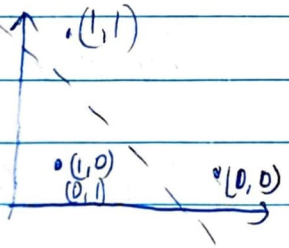
By applying non-linear hidden functions taking $t_1 (0,0)$ and $t_2 (1,1)$ as cluster/gaussian centres,

$$\phi_1(x) = \exp\left(-\|x - t_1\|^2\right)$$
$$\phi_2(x) = \exp\left(-\|x - t_2\|^2\right)$$

Original input (x_1, x_2) mapped to $(\phi_1(x), \phi_2(x))$

x (06)	$\phi_1(x)$	$\phi_2(x)$
(1, 1)	0.1353	1
(0, 1)	0.3678	0.3678
(0, 0)	1	0.1353
(1, 0)	0.3678	0.3678

As a result, patterns get reshaped into new shape, classes get rearranged and become separable by hyperplane.



★ Interpolation Problem (To prove linear solution exists)

Suppose we are given N input points $x_i \in R^{m_0}$, $i = 1, 2, \ldots, N$

Corresponding desired outputs $d_i \in R$

Then we need to find a function: $F: R^{m_0} \to R$ such that $F(x_i) = d_i \quad \forall i$

$$F(x) = \sum_{i=1}^{N} w_i \, \phi(\|x - x_i\|)$$

where $x_i \to$ centers of training points

$w_i \to$ unknown weights

$\phi \to$ RBF, $\|x - x_i\| \to$ distance

$$\sum_{i=1}^{N} w_i \, \phi(\|x_j - x_i\|) = d_j \quad \forall j$$

$\Phi \cdot w = d \to$ desired output

$\Phi \to$ RBF activation matrix $(N \times N)$

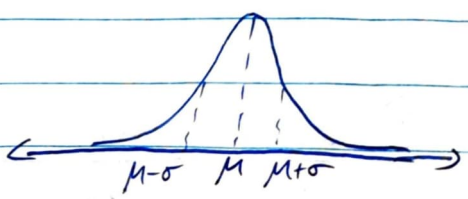
If ϕ is invertable, $w = \phi^{-1} d$

- Hidden layer size can equal no. of training samples
- Output layer becomes linear combination, and training reduces to solving linear equations

☆ Radial Basis Function (RBF) Networks

- Based on Cover's theorem and interpolation theory, it is a 3-layer feedforward neural network that uses non-linear radial functions in hidden layers and does not use backpropagation.
- The RBF hidden neuron measures how close the input is to it's center using Gaussian function.
- If input is close, that neuron activates strongly, else if input is far, neuron barely activates.
- Gaussian distribution is smooth, differentiable, localized, radially symmetric and mathematically convenient. (Bell-shaped curve)

$$\phi(x) = \exp\left(\frac{-\|x-c\|^2}{2\sigma^2}\right)$$



$c \rightarrow$ Gaussian center

$\sigma \rightarrow$ spread parameter

- Output of RBF hidden neuron depends on Gaussian function and distance from center.

- Output of RBF network is linear combination

$$\left(y(x) = \sum_{j=1}^{M} w_j \phi_j(x)\right)$$

M → no. of hidden layer neurons

- Gaussian is radially symmetric as it depends on distance from center, hence same output for all points having same distance from center $\|x - c_j\|$.

- Small σ → narrow peaks, localized, overfitting risk
  Large σ → wide smooth curves, better generalization but underfitting risk.

- RBF is powerful as : (a) Faster training
  (b) Good for function approximation
  (c) Hybrid learning (Sup + Unsup)
  (d) Strong theoretical base

- Two ways to train RBF network:

① Direct Weight Computation: Choose centers, fix σ, compute activation matrix ϕ

$$W = (\phi^T \phi)^{-1} \phi^T y$$

② Iterative Weight Update: Initialize weights randomly.
For each sample, $w_{j,new} = w_{j,old} + \eta \cdot (y_i - \hat{y}_i) \cdot \phi_j(x_i)$

- Here Input layer → Hidden Gaussian neurons → Linear output layer

☆ K-Means Clustering

- To choose centers c_j in real RBF networks.
- In interpolation case, each training point = a center
  Hidden neurons = N.
  But as $N \uparrow$, heavy computation.
  Hence we group data into clusters, then use cluster centroids as centers.
- Clustering: Unsupervised learning method to group similar data points together.
- Points in same cluster → close to each other
  Point in diff clusters → far apart.

1. Choose K initial centroids randomly.
2. Assign points to the nearest centroid using any distance metric
3. Update centroids
   New centroid = mean of all points in that cluster.
4. Repeat until no changes.

- Cost function, $J = \sum_{j=1}^{K} \sum_{x_i \in C_j} \|x_i - \mu_j\|^2$ (minimizes within-cluster variance)

where μ_j → centroid of cluster j (K-clusters)
C_j → cluster j

- After K-Means, $c_j = \mu_j$ (cluster centroid → Gaussian centre)

Hidden layer, $\phi_j(x) = \exp\left(\frac{-\|x - \mu_j\|^2}{2\sigma^2}\right)$

- Now from one neuron per training sample to one neuron per cluster, leading to smaller hidden layer and faster training.

★ Recursive Least Squares (RLS)

- How we train output layer of RBF network.

- After K-Means, we have centers c_j fixed, σ fixed, hidden activations $\phi_j(x)$ are known (fixed)

We need to find $W = [w_1, w_2, w_3, \ldots, w_M]^T$ such that $y(x) = \sum_{j=1}^{M} w_j \phi_j(x)$ matches desired output.

- We want to minimize $J = \Sigma(d_i - y_i)^2$ to determine $W = (\Phi^T \Phi)^{-1} \Phi^T d$
  But (a) matrix inversion is expensive
  (b) Not good for online learning.

- RLS is an efficient way to update weights recursively without recomputing everything.

- Let $W(n)$ → weight vector at step n
  $R(n)$ → correlation matrix
  $g(n)$ → gain vector (how much we should trust new data sample)
  $\alpha(n)$ → prior estimation error

- At each step, we measure prediction error and adjust

weights proportionally, based on gain vector.

① Compute Gain vector:

$$g(n) = \frac{P(n-1)\phi(n)}{1 + \phi^T(n) P(n-1) \phi(n)}$$

② Compute error:

$$\alpha(n) = d(n) - \phi^T(n) w(n-1)$$

③ Update weights:

$$w(n) = w(n-1) + g(n)\alpha(n)$$

④ Update inverse correlation matrix:

$$P(n) = P(n-1) - g(n)\phi^T(n) P(n-1)$$

☆ Hybrid Learning Procedure

— Train Hidden Layer (Unsupervised)

- Use K-means clustering to define hidden layer:

$$\phi_j(x) = \exp\left(\frac{-\|x - c_j\|^2}{2\sigma^2}\right)$$

— Train Output Layer (Supervised)

- Compute activation matrix ϕ
- Use Least Squares or RLS to determine w (ideal)

→ • Input → K-Means → Centers → Gaussian Activation → Output ← RLS ←

- However, there is no global optimality criterion combining both stages like MLP (hidden layer & output layer optimized separately)

☆ Self-Organizing Maps (SOM)

- An unsupervised neural network that maps high dimensional data onto low-dimensional grid while preserving topology.
- Similar inputs should activate nearby neurons.
- Similar inputs → nearby map nodes
  Dissimilar inputs → far apart nodes.
- SOM has only 2 layers:

(a) Input Layer (just distributes input in m neurons) (no computation)

(b) Output / Competitive Layer (neurons arranged in 1D/2D grid) (each neuron has weight vector, same dim as input)

— Learning

① Sampling
② Competition (Winner Selection)

- Each neuron in SOM has a weight vector:

$$W_j = [w_{j1}, w_{j2}, w_{j3}, \ldots, w_{jm}]$$

where j → neuron index, m → dimension of input

- All neurons compute distance for input x → $\|x - W_j\|$
- Best Matching Unit (BMU) represents the neuron with the

minimum distance from input.

$$i(x) = \arg\min_j \|x - w_j\|$$

② Cooperation (Neighborhood Concept)

- Not only winner updates, it's neighbours also update.
  Topology = relative arrangement / neighbourhood relationship of data points.

- Neighbourhood is based on physical location in grid, not weight similarity

- Hard Neighbourhood, $h_{j,i} = \begin{cases} 1 & \text{if within radius} \\ 0 & \text{otherwise} \end{cases}$
  (Only nearby neurons update)

- Gaussian Neighborhood, $h_{j,i} = \exp\left(\frac{-d_{j,i}^2}{2\sigma^2}\right)$
  (All neurons update, but far from centre update less)

- General update rule:

$$w_j(n+1) = w_j(n) + \eta(n)\, h_{j,i}(n)\,(x(n) - w_j(n))$$

↳ neighbourhood function

Here $\eta(n)$ → learning rate changes over time

$$\eta(n) = \eta_0 e^{-n/T}$$

Large → fast learning
Small → stable convergence.

RECURRENT NEURAL NETWORKS

- Neural network where outputs are fed back into the network as inputs.

- Unlike feedforward networks like MLP, RNNs have feedback loops which allow them to remember past information, making them suitable for sequential or time-dependent data.

- Output at time t_3 depends on the output at time t_1 and t_2, since the network stores state information.

- A dynamically-driven RNN receives external inputs over time, state evolves dynamically and outputs depend on both current input and previous state.

- Feedback is the core idea of RNNs and connections can occur between (a) Hidden ➡ Input and (b) Output ➡ Hidden

- State of a system summarizes all past information needed to predict the future.

Mathematically, state vectors (x_n), input vectors (u_n) output vectors (y_n)

$x_{n+1} = f(x_n, u_n)$, $y_n = g(x_n)$

(Next state depends on current state + input) (Output depends on current state)

- RNNs are powerful as they can model dynamic systems. A few applications include:

(i) Non-linear prediction (stock prices, weather)
(ii) Communication systems (adaptive equalization of communication channels)
(iii) Speech processing (voice assistants)
(iv) Time series modelling (financial data, sensor signal

- Main drawback of feedforward networks is lack of memory of past inputs during processing.

- RNNs can model temporal dependencies, on the other hand.

Eg — In speech recognition of HELLO,
H → E → L → L → O
Model must remember previous sounds.

- RNNs operate in discrete time steps and each step depends on the previous state.

u_1 → network → y_1
u_2 → network → y_2
u_3 → network → y_3

— Characteristics of RNN

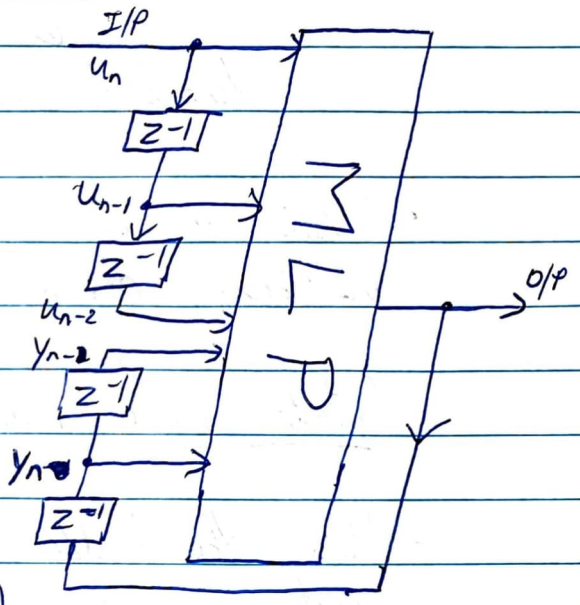
- Feedback connections (Outputs influence future inputs)
- Internal state (memory) (network remembers past information)

- Dynamic behaviour (output changes over time)
- Ability to model non-linear ~~activation function~~ systems (due to non-linear activation functions)

☆ Architectures

- Built using non-linear mapping ability of MLPs and introduces feedback loops to handle temporal data

– Input-Output Recurrent Model



- Comprises of input signal, multilayer perceptron, output feedback and delay units (z^{-1}) (memory)
- Input and output are both passed through tapped delay lines before entering the network (stores past values)
- Neural networks normally do not remember past values ($u_n \rightarrow MLP \rightarrow y_n$), as a result, delay blocks (z^{-1}) shift the signal one time step back, enabling the network to see a window of time instead of one time point (NOW)
- As a result, the MLP receives a data window containing present and past input values and delayed output values, as an input vector.

$I/P \Rightarrow [u_n, u_{n-1}, \ldots, y_n, y_{n-1}]$

- The MLP produces the output y_{n+1} (prediction) with the help of the non-linear function learned by the neural network (F)

$$y_{n+1} = F(y_n, y_{n-1}, \ldots, u_n, u_{n-1}, \ldots)$$

- Such a model architecture is called NARX (Non-linear AutoRegressive model with exogenous inputs)
  - depends on past outputs
  - external inputs influence output.
- NARX models are widely used in time-series prediction, control systems, signal processing.

– State-Space Model

- Here the hidden layer represents the state of the system, whose output is fed back into the input through delay units.
- In NARX, memory was created using explicit delay blocks (outside the neural network)
- Instead of storing all the past inputs and outputs, this network stores states (internal memory of the system) which stores all information needed about the past (summary, compressed memory)

State equation: $x_{n+1} = \alpha(x_n, u_n)$ (hidden state, input; α → non-linear hidden layer function)

Output: $y_n = B x_n$ (B → weight matrix of output layer)

- Hidden layer → non-linear
  Output layer → linear, thus allowing the network to model complex non-linear systems

— Recurrent MLP

- Contains multiple hidden layers and feedback connections in each layer through delay units.

Let $x_{1,n}$ and $x_{2,n}$ be the hidden layer outputs,

$$x_{1,n+1} = \phi_1(x_{1,n}, u_n)$$
$$x_{2,n+1} = \phi_2(x_{2,n}, x_{1,n+1})$$

$$x_{0,n+1} = \phi_0(x_{0,n}, x_{k,n+1})$$ (k → no. of hidden layers)

— Second-Order Network

- Normally a neuron computes:

$$v_k = \sum w_{kj} x_j + \sum w_{ki} x_i$$

Here weights multiply inputs individually

- In second-order networks, inputs are multiplied together

$$v_k = \sum\sum w_{kij} x_i u_j$$ (neuron responds to product of inputs)

- Similar to Deterministic Finite Automata (DFA) which is a system with finite states and state transitions based on inputs
(current state + input → next state) $\delta(x_i, u_j) = x_k$

- First-Order neurons treat state and input seperately cannot naturally detect a specific pair.
On the other hand, in Second-Order neurons, the neuron activates only when both conditions occurs together.

Ex: Waiting → $x_1 = 1$ Inputs ⇒ coin → u_1
CoinInserted → $x_2 = 1$ button → u_2
Dispense → $x_3 = 1$ done → u_3
(Only one state neuron is active at a time)

state = Waiting $[1\ 0\ 0]$ (Waiting & Coin)
input = Coin $[1\ 0\ 0]$
Neuron activates → $g(\text{Waiting, Coin}) \rightarrow \text{CoinInserted}$
$(x_1, u_1) \rightarrow x_2$
state = CoinInserted $[0\ 1\ 0]$
input = Done $[0\ 0\ 1]$
Neuron stays off (state × input = 0)

✱ Universal Approximation Theorem

- A neural network with one hidden layer and non-linear activation function can approximate any continuous function to any desired accuracy, provided it has enough neurons.

- Neural networks are very powerful function approximators

Ex: Dynamic systems follow equations like:

$$x_{n+1} = f(x_n, u_n)$$
$$y_n = g(x_n)$$

where $x_n \rightarrow$ system state, $u_n \rightarrow$ input, $y_n \rightarrow$ output

A neural network can approximate f and g with arbitrary accuracy.

- An RNN can model the behaviour of dynamic systems by approximating the underlying equations.

- Accuracy improves with more hidden neurons.

✱ Controllability and Observability

- A system is <u>controllable</u> if we can move the system from any initial state to any desired state using inputs within a finite time.

- In RNN, state of network is x_n (hidden layer output) and input is u_n (external signal)
If by changing u_n we can make the network

reach any state x_n, network is said to be controllable

Ex: Think of driving a car:

States: position, speed, direction
Inputs: accelerator, brake, steering

If you can drive the car from any position to any other position, the system is controllable

- A system is observable if we can determine the internal state of the system by looking at inputs and outputs

- In RNN, hidden layer outputs represent the state (x_n) and the output produced → y_n.
If internal state x_n can be determined from input-output behavior, the network is observable.

Ex: Imagine a car inside a box, you cannot see the car directly
But you observe steering input, speed output
From those measurements, you might figure out the current position and velocity of the car

- If a RNN is controllable, we can guide the network to desired internal states.
If a RNN is observable, we can understand what the network is doing internally.

- These properties help analyze stability and learning behaviors.

- Local controllability means controllability near an equilibrium point (origin) / around a specific state.

- Local observability means determining the state near an equilibrium point from outputs.

- Equilibrium state is where system does not change ($x_{n+1} = x_n$)

Ex: State space equations: $x_{n+1} = f(x_n, u_n)$ / $y_n = g(x_n)$
Controllability → ability to move x_n using u_n
Observability → ability to infer x_n using y_n

☆ Computation Power of RNNs

- Refers to what kind of problems a model can solve, what type of patterns it can recognize and how complex are the systems it can simulate.

- RNNs have great computational power than feed-forward networks because their feedback connections allow them to store state information and process sequential data, enabling them to simulate finite automata and model dynamic systems

- Feedback connections, memory & state representation

allow RNNs to process sequences and time-dependent data, which feedforward networks cannot do easily as they do not remember past inputs.

- Earlier research used neurons with hard threshold activation functions that behave like binary logic units (0 or 1) instead of smooth functions like sigmoid
- RNNs can process sequential patterns by simulating a finite automaton that changes state based on inputs where the state stores progress through the sequence

— Siegelmann's Theorems

I) All Turing machines can be simulated by fully connected recurrent networks with sigmoidal activation functions

- A Turing machine is the most general model of computation, which when simulated, can compute any computable function.
Hence RNNs have very high computational power

- Turing machine consist of:

(i) Control Unit (maintains current state of the machine)
(ii) Linear Tape (infinite memory divided into cells, each storing a symbol)
(iii) Read-Write Head (moves left/right on the tape, reading or writing symbols.

II) A NARX model/network with one hidden layer, Bounded One-Sided Saturation (BOSS) activation functions and a linear output neuron can simulate a fully connected RNN, except taking slightly longer to compute.

- If a fully-connected RNN solves a task in time T, the equivalent NARX network takes $(N+1)T$ where N = no. of neurons.

- BOSS activation function must satisfy 3 conditions:

(i) Bounded Range $(a \leq \phi(x) \leq b)$

(ii) Saturation on one side $(\phi(x) = \text{constant for } x \leq s)$

(iii) Non-constant function (changes with input)

✱ Learning Algorithms

— Training Methods for MLP

Feature	Batch Mode	Stochastic (Sequential)
• Gradient Calculation	Uses entire dataset	Uses one sample at a time
• Parameter Update	Updated once per epoch	Updated after every sample
• Speed of Learning	Slower	Faster
• Stability	Stable, smooth updates	Unstable, noisy updates
• Convergence	Accurate but slow	Fast but may oscillate
• Memory Requirement	High (stores full dataset)	Low (one sample only)

• Computation	Good for parallel processing	Simple, but less efficient overall
• Behaviour	Moves in smooth direction	Moves in zig zag path
• Best-Use Case	Large datasets, offline training	Real time, streaming data
• Example	Image classification (CNNs)	Stock prediction, online learning

— Epoch-wise Training

• Training the RNN using one complete sequence of temporally ordered input-target pairs, after which the weights are updated and network state is reset.

① Input a full sequence
② Run network from start to end
③ Compute total error
④ Update weights
⑤ Reset network
⑥ Repeat for next sequence

• Unlike MLPs where each sample is independent, in RNNs each sequence forms one training sample due to temporal dependencies.

• In MLP, 100 rows ⇒ 1 epoch = all 100 rows
In RNN, 100 sequences ⇒ 1 epoch = all 100 sequences.
Output depends on past inputs.

- MLP thinks each row is independent

Hours	Result	
2	Fail	(2 hours today has nothing to do with yesterday)
5	Pass	(1 row = 1 sample)
8	Pass	(Update weights after feeding each row)

- RNN tries to understand the word HELLO in a sequence.
H → E → L → L → O (meaning comes from full sequence, not individual pieces)
(HELLO = one sample)

- First start fresh (brain is empty), then feed sequence
Input H → remembers (network is building memory as it goes)
Input E → update memory
Input L → update
Input L → update
Input O → update
Then check if it predicted correctly. If yes, update weights after learning from full sequence.
Then reset memory and start next sequence fresh.

- For example, when watching movies (sequence), scene = time step, you do not judge a movie from one scene, you have to watch the WHOLE movie to decide.
Watch full movie → give review → forget → next movie

— Continuous Training

- Recurrent network updates its weights in real time as new data arrives, without resetting the network state.

- Suitable for real-time applications such as speech processing and sensor data.

Input at t_1 —predict→ update weights
Input at t_2 → update weights
Input at t_3 → update weights
⋮ (never stops)

Feature	Epochwise Training	Continuous Training
Data	Fixed sequences	Continuous stream
Update	After sequence	After each time step
Reset	Yes	No
Learning	Offline	Online
Stability	More stable	Less stable
Memory	Higher	Lower
Problem	Slow	Forgetting issue

Note: In RNNs at each time step,
① Read input → update memory

$h_t = f(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1})$

h_t → memory (hidden state)
x_t → input at time t
(use old memory + new input)
(repeat)
(network is carrying memory forward)

$y_t = W_{hy} \cdot h_t$

✱ Backpropagation Through Time (BPTT)

- The issue with RNNs is that same weights are used across time. Output at time t depends on all previous states

- BPTT solves this issue by converting RNN into a deep feedforward network by unfolding it through time, then apply standard backpropagation through entire sequence

① Unfold the RNN (each time step = one layer)

$x1 \to h1 \to y1$ (same weights everywhere)

$x2 \to h2 \to y2$

$\vdots \quad \vdots \quad \vdots$ (deep network depth = sequence length)

$xT \to hT \to yT$

② Forward Pass

For each time step:

$h_t = f(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1})$

$y_t = g(W_{hy} \cdot h_t)$

Compute error: $E_t = Loss(y_t, target_t)$

Total error: $E = \sum_{t=1}^{T} E_t$

③ Backward Pass

Here we compute gradient $\frac{\partial E}{\partial W}$ however each weight affects multiple time steps. $\frac{\partial E}{\partial W} = \sum \frac{\partial E_t}{\partial W}$

According to chain rule, say dependency of E_3 sequence

$E_3 \to h_3 \to h_2 \to h_1 \to W$

(Gradients flow backward through time)

$$\frac{\partial E_3}{\partial W} = \left(\frac{\partial E_3}{\partial h_3}\right) * \left(\frac{\partial h_3}{\partial h_2}\right) * \left(\frac{\partial h_2}{\partial h_1}\right) * \left(\frac{\partial h_1}{\partial W}\right)$$

For recurrent weights W_{hh}

$$\frac{\partial E}{\partial W_{hh}} = \sum_t \sum_k \left(\frac{\partial E_t}{\partial h_t}\right)\left(\frac{\partial h_t}{\partial h_k}\right)\left(\frac{\partial h_k}{\partial W_{hh}}\right)$$

Using this, update weights through regular weight update formula.

Ex:- You write a sentence word by word.
At the end, teacher gives feedback and you go back word by word to adjust earlier mistakes.

- Cons: Vanishing Gradient (gradient approach 0, learning nothing) and Exploding Gradient (training unstable)

- Can be solved using Gradient Clipping (limit gradient size), better activation functions like ReLU variants and LSTM / GRU to control memory flow.

- Truncated BPTT is useful for reducing memory, faster training and less vanishing problem. Instead of full sequence it uses small chunks of the data, then updates weights
  x1 - x10 → update
  x11 → x20 → update

- Time Complexity : O(T)
  Space Complexity / Memory : O(T × hidden_size)

- Offline, batch training on fixed dataset, sequence known fully. (Ex:- Machine translation, language modelling)

$$\frac{\partial E_3}{\partial W} = \left(\frac{\partial E_3}{\partial h_3}\right) * \left(\frac{\partial h_3}{\partial h_2}\right) * \left(\frac{\partial h_2}{\partial h_1}\right) * \left(\frac{\partial h_1}{\partial W}\right)$$

For recurrent weights W_{hh}

$$\frac{\partial E}{\partial W_{hh}} = \sum_t \sum_k \left(\frac{\partial E_t}{\partial h_t}\right)\left(\frac{\partial h_t}{\partial h_k}\right)\left(\frac{\partial h_k}{\partial W_{hh}}\right)$$

Using this, update weights through regular weight update formula.

Ex:- You write a sentence word by word.
At the end, teacher gives feedback and you go back word by word to adjust earlier mistakes

- Cons: Vanishing Gradient (gradients approach 0, learning nothing) and Exploding Gradient (training unstable)

- Can be solved using Gradient Clipping (limit gradient size), better activation functions like ReLU variants and LSTM / GRU to control memory flow.

- Truncated BPTT is useful for reducing memory, faster training and less vanishing problem. Instead of full sequence it uses small chunks of the data, then updates weights
  x1 - x10 → update
  x11 → x20 → update

- Time Complexity : O(T)
  Space Complexity / Memory : O(T x hidden_size)

- Offline, batch training on fixed dataset, sequence known fully. (Ex:- Machine translation, language modelling)

- Cannot handle infinite streams, memory intensive, slow for long sequences.

☆ Real-Time Recurrent Learning (RTRL)

- Used to train RNNs in real time. Computes gradients forward in time by maintaining the sensitivity of hidden states with respect to weights, updating at each time step.

- Instead of unfolding the network like in BPTT, we compute $\frac{\partial E_t}{\partial W}$ at each time step.

- BPTT = backward flow of error
  RTRL = forward tracking of influence, keeping track of how each weight affects the current hidden state

- $P_t = \frac{\partial h_t}{\partial W}$ (sensitivity of hidden state to weights)

At time t, $P_t = \left(\frac{\partial h_t}{\partial h_{t-1}}\right) P_{t-1} + \frac{\partial h_t}{\partial W}$

(New influence = Old influence passed forward + Current direct effect)

- At each time step,

$$\frac{\partial E_t}{\partial W} = \left(\frac{\partial E_t}{\partial h_t}\right) \times P_t$$

$$W = W - \eta \left(\frac{\partial E_t}{\partial W}\right)$$

Ex: • You are writing an essay and correcting mistakes at the same ~~mistake~~ time.

Cons: For each weight, we track $\frac{\partial h_i}{\partial W_j}$

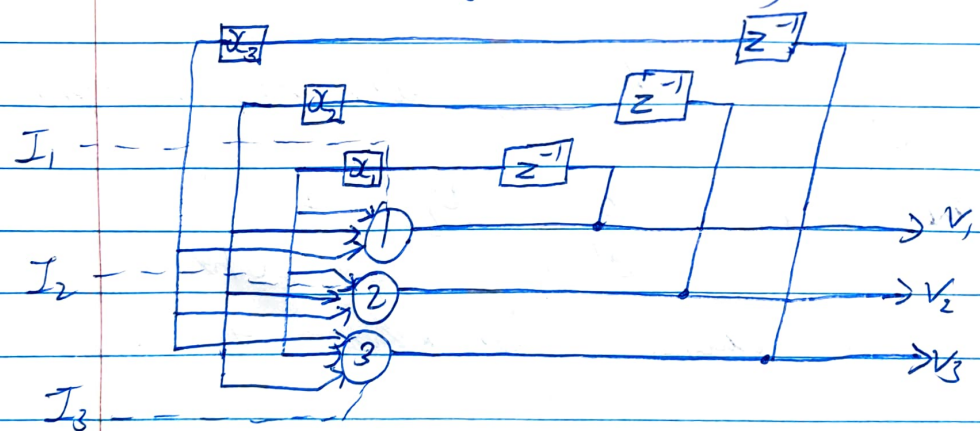
If N neurons exist, total derivatives = $N \times N \times N = O(N^3)$
Total complexity = $O(N^4)$
(Computation explodes VERY FAST)
(However, memory usage is low due to online learning)

- Useful for real-time learning, streaming data and when no reset available.

Eg:- Robotics control, online prediction, adaptive systems.

☆ Hopfield Network

- A type of RNN that has global feedback (every neuron connected to every other neuron)



- Single layer network that is fully connected (each neuron connected to all other neurons and itself too)
No seperate hidden/output layers. like in MLP.

- Dynamic behaviour → output evolves over time
Associative memory → can recall stored patterns

Stores information and Hardware-friendly (analog implementation possible)

- Each neuron follows:

Internal state: $\dot{u}_j(t) = -\eta u_j(t) + \sum_{i=1}^{n} W_{ji} V_i(t) + I_j$

Output: $V_j(t) = g(u_j(t))$

where $u_j(t)$ → internal state η → ~~bias~~ decay factor
$V_j(t)$ → output of neuron g → activation function
W_{ji} → weight from neuron $i \to j$
I_j → external input (bias)

1. Apply input vector x
2. Compute output V
3. Feed output back as new input ($x \leftarrow V$)
4. Repeat until output stops changing (settles / converges)

- Iteratively, $t=0$: input → output
  $t=1$: previous output → new output
  $t=2$: continues
  Until $V(t) = V(t-1)$ (Stable state reached)
  (Memory recall)

- Discrete: $u_j(k) = \sum W_{ji} V_i(k-1) + I_j$
  $V_j(k) = g(u_j(k))$

- Energy function describes how stable the network is (how well current state matches stored pattern)

$$E = -\frac{1}{2} \sum_i \sum_j W_{ij} V_i V_j + \sum_i \theta_i V_i$$

(Lyapunov Function) θ_i V_i → bias

- If neurons align with weights → energy ↓
  Neuron mismatch → energy ↑

- Energy derivative, $\frac{dE}{dt} = -\sum (u_j(t))^2 \cdot \left(\frac{dV_j(t)}{du_j(t)}\right)$
  always decreases or stays same
  squared rate of change of internal activation
  activation function derivative (+ if monotonic)

- Stability Conditions for energy to decrease include:
  (a) Symmetric weights : $W_{ij} = W_{ji}$
  (b) No self loops : $W_{ii} = 0$
  (c) Activation function should be monotonically increasing (sigmoid, tanh)

— Associative Memory

- Memory system that retrieves full data from partial/noisy input (fills in the missing parts)

Ex- Stored pattern : 101010 (stable state) (energy minima)
Noisy Input given : 100000
Output : 101010 (through recall process)

- Hopfield does not store addresses, it stores patterns in weights (Content - Addressable Memory (CAM))

- Stores patterns in a vector of size N.
  Bipolar : {-1, +1}
  Binary : {0, 1} (not preferred)

(must be symmetric)

Weights are computed using outer product of patterns.

For storing p patterns: $W = \sum_{k=1}^{p} x^k (x^k)^T$, $W_{ii} = 0$

(matrix = pattern × transpose)

Patterns became energy mining which network converges to.

Work well for small number of patterns only.

- Pseudoinverse approach is more accurate, less interference, removes overlap between patterns, though computationally expensive

$$W = X(X^T X)^{-1} X^T$$

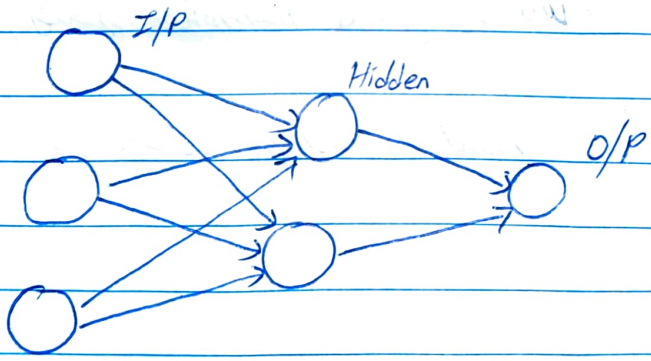
- For N neuron, max patterns we can store, $p_{max} \approx 0.138N$ as too many patterns can cause interference, energy landscape can get messy, could converge to wrong pattern, produce spurious (fake memories) states, mix / noise sensitivity.

- Initially, (noisy) input should be close to stored pattern.

—x—

FEEDFORWARD NETWORK NETWORKS

- Most basic and important deep learning models used for image classification, spam detection, regression, etc.
- Goal is to approximate some function $y = f(x; \theta)$ where x → input (data), y → output (prediction), θ → weights + biases (to learn mapping from input to output)
- Feedforward as data flows in only one direction, no feedback like recurrent networks.
- Represented by composing many functions (network = chain of functions)
  $f(x) = f^{(L)}(f^{(L-1)}(\ldots f^{(1)}(x)))$
  Each layer = function.



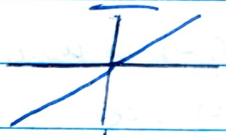
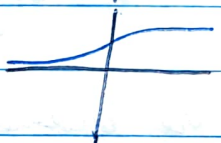
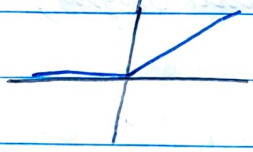
- Input Layer: Takes raw data (image → pixels, dataset → features)
- Hidden Layers:
  - Allow model to learn patterns, relationships.
  - Perform transformations.
  - Not directly supervised as training data does not show desired output for hidden layers. Hence, model learns hidden layers by itself.

Ex: Image: Layer 1 → edges
Layer 2 → shapes
Layer 3 → objects

- Depth = number of layers (more depth → more complex learning)
  Width = number of neurons per layer (more feature capacity)

- Output Layer: Produces final result.

- Activation Functions: Without it, model becomes linear, cannot learn complex patterns.

	Name	Plot	Function	Derivative	Range
①	Identity		$f(x) = x$	1	$(-\infty, \infty)$
②	Binary Step		$f(x) = \begin{cases} 0, & x<0 \\ 1, & x \geq 0 \end{cases}$	$\begin{cases} 0, & x \neq 0 \\ \text{undefined}, & x=0 \end{cases}$	$\{0,1\}$
③	Sigmoid / Logistic		$f(x) = \frac{1}{1+e^{-x}}$	$f(x)(1-f(x))$	$(0,1)$
④	Tanh		$f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	$1-f(x)^2$	$(-1,1)$
⑤	ReLU		$f(x) = \max(0,x)$	$\begin{cases} 0, & x<0 \\ 1, & x>0 \\ \text{undefined}, & x=0 \end{cases}$	$[0,\infty)$

Note: Identity ⇒ no change (linear)
Binary Step ⇒ used in perceptron
Sigmoid ⇒ used for binary classification
Tanh ⇒ Centered output
ReLU ⇒ fast, used for deep learning, solves vanishing gradient problem.
Softmax ⇒ multiclass classification.
(converts outputs → probabilities)

- Working of Multi-Layer Feedforward NN:

① Forward Propagation

- Each neuron recieves input, multiplies it with corresponding weights, adds bias and applies an activation function. Output of one layer becomes input of next layer.

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)} \qquad l \text{ - s layer number}$$
$$a^{(l)} = f(z^{(l)})$$

② Weight Initialization : Proper initialization helps for faster convergence.

If weights are too large → exploding gradients
too small → vanishing gradients

③ Training Process: Weights are updated using gradient descent to minimize loss (error between predicted and actual output)

- Forward Pass → Compute Error → Backward Pass → Update Weights → Repeat.

- Forward pass only gives output, it does not tell which weight is wrong and how to fix it.
Backpropagation solves this by computing gradients and updating weights accordingly

- Gradient = direction of steepest increase, computed

chain rule, $\frac{\partial L}{\partial w}$ (how much loss changes if weight changes)

If weight increases loss → decrease it
If weight decreases loss → increase it

$$\left(W_{new} = W_{old} - \eta \frac{\partial L}{\partial w}\right)$$

where η → learning rate (controls step size)
[too high → overshooting
too low → slow learning]

Goal is to move weights in direction of minimum loss.

Pros: Efficient computation, works for deep networks, scales well
Cons: Vanishing gradient (gradient too small, slow learning)
Exploding gradient (gradients too large, unstable training)
[Can be solved using ReLU, proper initialization & optimization]

☆ Pattern Classification using Perceptron

- Perceptron is the simplest type of ANN used for binary classification, using threshold (step) activation and works only on linear seperable data (AND/OR) not XOR.

- Input Layer: $x_1, x_2, x_3, \ldots, x_n$
Weights: $w_1, w_2, w_3, \ldots, w_n$
Summation function, $z = \sum w_i x_i + b$
Activation function produces final output, $y = f(z)$

$$f(x) = \begin{cases} +1, & x > \theta \\ 0, & -\theta \le x \le \theta \\ -1, & x < -\theta \end{cases}$$

Weight Update: $w_{new} = w_{old} + \eta \times error \times x$

$b_{new} = b_{old} + \eta \times error$

If prediction is wrong → adjust weights, else do nothing

- Multiclass Classification: One input → one class
  Output = one-hot vector [0 1 0]
  Softmax activation
- Multi-label Classification: One input → multiple classes
  Output = multi-hot vector [1 0 1]
  Sigmoid activation

☆ Gradient-Based Learning

- Goal: $\min_{\theta} L(\theta)$

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots\right), \quad \nabla L(w) = \frac{\partial L}{\partial w}$$

$$w = w - \eta \nabla L(w)$$

- Types of Gradient Descent:

① Batch Gradient Descent: Uses entire dataset
Accurate but slow

② Stochastic Gradient Descent: Uses 1 sample at a time
Fast but noisy

③ Mini-Batch Gradient Descent: Uses small batch
Best balance

- Optimization Techniques:

① Momentum: Use previous gradients, $v = \beta v + \nabla L$
Helps in faster convergence, smooth updates

② AdaGrad: Different learning rate for each parameter

③ RMS Prop : Fixes Adagrad issues
④ Adam : Combines Momentum + RMSProp
⑤ Gradient Clipping : Limit gradient size to prevent exploding gradients

Empirical Risk Minimization (ERM)

- Principle where a model is trained to minimize average loss over a dataset, instead of minimizing error on one sample.

$$R(W) = \frac{1}{n} \sum_{i=1}^{n} L(y_i, f(x_i))$$

($f(x_i)$ → predicted output)

where $R(W)$ → empirical risk, n → no. of samples.
L → loss function, y_i → actual output

- Goal : Find $w^* = \text{argmin } R(w)$
Update : $w_{new} = w_{old} - \eta \nabla_w R_n(w)$

- Loss function used include :

① Mean Squared Error : For regression

$$L(y, \hat{y}) = \frac{1}{n} \Sigma (y - \hat{y})^2$$

② Binary Cross-Entropy : $L(y, \hat{y}) = \frac{-1}{n} \Sigma [y \log \hat{y} + (1-y) \log (1-\hat{y})]$

③ Categorical Cross-Entropy : For multiclass classification

$$L(y, \hat{y}) = \frac{-1}{n} \Sigma \, y \log(\hat{y})$$

Regularization

- Any modification to a learning algorithm intended to reduce generalization error (test error), not training error.
- Makes model perform well on new/unseen data to improve generalization by not memorizing training data (overfitting)
- Bias = error due to simplification (underfitting)
  Variance = error due to sensitivity to data (overfitting)
  Regularization trades increased bias for reduced variance to produce slightly simpler model + much better generalization
- Either add constraints on parameters, or add penalty term in loss function

$$L'(w) = L(w) + \lambda \Omega(w)$$

(original loss) ↑ regularization strength → penalty

- Parameter Norm Penalties prevent weights from becoming too large.
- L2 / Ridge Regularization: $\Omega(w) = \frac{1}{2} \sum w_i^2$

  Shrinks weights smoothly, keeping all features, encourages small weights, thus stabilising model.
- L1 / Lasso Regularization: $\Omega(w) = \sum |w_i|$

  Force some weights = 0 (sparsity), performs feature selection
- Weight Update: $\hat{w} = w - \eta \left(\frac{\partial L}{\partial w} + \lambda w \right)$

☆ Dataset Augmentation

- Technique where we take existing data, apply transformations and generate new synthetic samples
(Create fake data and add it to the training set)

- Improves generalization (model performs well on unseen data)
- Reduces overfitting (model doesn't memorize training data)
- Handles Class Imbalance (increase samples of minority class)
- Makes Model Robust (handles real-world variations)

- However augmentation should not change meaning of data

Ex: Digit 6 → rotate → becomes 9 (wrong)

- Dataset Augmentation techniques on image data include:

Geometric Transformation: Rotation ↻ Flipping ↔ ↕
Scaling/Cropping, Translation ↔ ↕

Color & Lighting: Brightness, Contrast (lighting)
Saturation, Hue (color)
Gaussian Noise (random noise)

☆ Noise Robustness

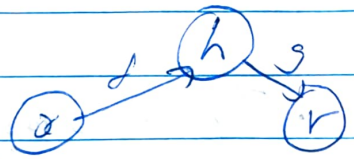
- Adding noise to inputs can act as a form of regularization
Train model to handle imperfect noisy data.

- Noise can be added to inputs or hidden layers to force deeper feature learning.
- Types of noise include Gaussian noise, Dropout (randomly removing neurons)
Salt & Pepper Noise

☆ Autoencoders

- Neural networks that learn to encode input data into a compressed latent representation and then reconstruct it.

- Encoder compresses input into latent representation, $h = f(x)$
  Decoder reconstructs input, $r = g(h)$
  Finally, $\hat{x} \approx x$



- Latent variables are hidden representations that can be inferred indirectly through a model

- Autoencoders are trained using backpropagation, gradient descent and mini-batch learning technique.

- Goal: Minimize reconstruction error, $L = \|x - \hat{x}\|^2$

- Purpose: Dimensionality Reduction, Denoising, Data Compression, Feature Learning.

Ex: Input (x) → brain MRI image
(raw, high dimensional data, hard to directly understand patterns)
Encoder (Compression Step) → converts input into compressed representation h containing latent variables (hidden features that model discovers automatically) (cannot be directly observed but learned from data)
h = [Brain shrinkage level, ventricle size, texture abnormality, disease severity]

As a result, encoder converts complex image → meaningful features, removing noise and keeping only important info.

Decoder (Reconstruction) → Takes h and reconstructs image
$\hat{x} = g(h)$ such that Output ≈ Input

From the hidden latent features, model is also indirectly learning real-world causes such as disease progression, neuron degeneration rate and inflammation patterns, that drive the image appearance.

① Undercomplete Autoencoder: Hidden layer smaller than input layer, forcing model to learn important features only, thus preventing overfitting. Similar to PCA.

② Regularized Autoencoders: Add constraints to avoid trivial copying

(a) Sparse Autoencoder: By applying L1 regularization, only few neurons activated to learn only meaningful features (Works even with large hidden layer)

(b) Denoising Autoencoder: (Noisy Input → Clean Output)
Corrupting the input then learning to reconstruct the clean input. Prevents memorization, Robust feature learning

→

Type	Purpose	Key Mechanism	Use Case
Undercomplete	Feature compression	Bottleneck layer	Dimensionality reduction
Sparse	Feature extraction	L1 reg.	NLP, image proc.
Denoising	Robust rep. learning	Add noise + reconstruct	Img denoising, data recovery
Regularized	Prevent overfitting	Any additional constraint (sparsity, noise)	General DL tasks

DEEP NEURAL NETWORKS & CNNs

- DNN is an artificial neural network with multiple hidden layers between input and output, such that each layer learns increasingly abstract features.

- Key Components of DNNs:

(i) Input Layer (recieves raw data) (images, text, numerical data)
(ii) Hidden Layers (computations occur, extracts higher-level features)
(iii) Activation Functions (introduce non-linearity to model complex relationships)
(iv) Weights & Biases (trainable parameters that adjust during learning to minimize error)
(v) Output Layer (produces final prediction / classification)

- Training Process:

① Forward Propagation: Input → Hidden Layers → Output
Only compute predictions, no learning yet.
② Loss Function: Measures errors between predicted and actual output (MSE, Cross-Entropy)
③ Backpropagation: Using chain rule, compute gradients $\frac{\partial L}{\partial W}$ (how much should each weight change based on how wrong the model predicted)
④ Optimization Algorithms: Update weights using algorithms such as SGD, Adam.

$$W = W - \eta \frac{\partial L}{\partial W}$$

- Types of DNNs:

① Feedforward NN (basic, one-direction flow)
② Convolution NN (used for images, uses filters instead of full connections)

③ Recurrent NN (has memory) (used for sequences, time-series data)
④ Transformers (modern NLP tools like BERT) (use attention mechanism)

- DNN is basically a function $y = f(x; W, b)$
  Goal: training to find best W, b such that loss → minimum

Cons ① • However as network get deeper, errors have to travel back through many layers; more layers = more parameters = more complexity causing instability + inefficiency.

- During backpropagation: $\frac{\partial L}{\partial W}$ = chain of many derivatives (multiplied)

  Vanishing Gradient ⇒ gradient becomes very small ≈ 0 thus stopping learning (sigmoid, tanh)

  Exploding Gradient ⇒ gradient become very large, weights update too aggressively, training unstable

  Solutions: (i) Use ReLU (keeps gradients alive)
  (ii) Batch Normalization (prevent extreme values)
  (iii) Proper Initialization (Xavier/He) (keep σ^2 controlled)

② • Overfitting. Since DNNs have many parameters, they can memorize training data and perform poorly on unseen data
  Poor generalization, model learns noise instead of patterns

  Solutions: (i) Regularization (L1/L2) (penalize large weights)
  (ii) Dropout (randomly deactivate neurons)
  (iii) Data Augmentation (create more data, rotate/flip)
  (iv) Early Stopping (stop training before overfitting starts)

(3) Computational Complexity: Training DNNs require huge computation, large matrix math (slow + expensive)
Solutions: (i) Use GPU/TPU for parallel computation
(ii) Model Optimization (remove unnecessary layers)
(iii) Pruning (remove less important weights)
(iv) Quantization (reduce precision (float → int))

(4) Data Requirements: DNNs need large, high-quality datasets
Since more parameters ⇒ more data needed
Solutions: (i) Transfer Learning (use pre-trained models)
(ii) Data Augmentation (increase dataset artificially)
(iii) Semi-Supervised Learning (use a combo of labelled + unlabelled data)

(5) Hyperparameter Tuning: Choosing learning rate, batch size, number of layers, etc. based on trial-and-error.
Small change → huge effect
Solutions: (i) Grid Search (try all combos)
(ii) Random Search (try random combos)
(iii) AutoML (automates tuning)
(iv) Bayesian Optimization (probability-based)
(v) Validation Metrics Monitoring

(6) Long-Training Time: DNNs using large datasets can take hours and days to train.
Solutions: (i) Mini-Batch Training (faster than full dataset)
(ii) Distributed Learning (on multiple GPUs/TPUs)
(iii) Learning Rate Scheduling (adjust LR during training)

☆ Greedy Layer-Wise Training

- Train a DNN one layer at a time instead of all layers together - (sequentially)

① Train First Layer (Input ⇒ First hidden layer)
(Since input of hidden layer h_1 is input data (x), create a dummy output layer either acting as an Autoencoder (learn to reconstruct input) or Restricted Boltzmann Machine (RBM) to learn probability distribution.) (After learning, the parameters / weights of that layer can be assigned as input for next layer)

② Freeze first layer, Train Second
(Input = output of h_1, try to retrieve input and assign weights to h_2)

③ Repeat for all layers.

④ After all layers are trained, Train the entire network together using backpropagation to adjust all weights jointly.

Pros:
- Solves Vanishing Gradient problem (each layer trained independently so gradients need not travel through entire network)
- Better Weight Initialization (pretrained meaningful weights)
- Efficient Learning
- Works well for unsupervised learning (learns features w/o labels first)

Cons:
- Takes Longer (Multi-Stage Training)
- Less used Today (better methods such as Adam and Batch Normalization exist)

Ex: Instead of learning whole syllabus in one day.
Learn Chapter 1 → Chapter 2 → Chapter 3
Finally revise everything

★ Optimization for Training DNNs

① Stochastic Gradient Descent (SGD)

- An optimization algorithm used to minimize the loss function by iteratively updating parameters (per sample)

(i) Initialize parameters (random weights θ)
(ii) Compute Loss $L(\theta)$ (error between predicted & actual)
(iii) Compute Gradient $\nabla L(\theta)$ using backpropagation
(iv) Update parameters $\theta = \theta - \eta \nabla L(\theta)$
(v) Repeat until convergence (min (loss) (or) stopping condition reached)

- Mini-batch SGD uses a ~~sample~~ batch of m samples instead of full dataset

$$g \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L\left(f(x_i; \theta), y_i\right)$$

Faster but noisy
+ efficient

$$\theta \leftarrow \theta - \eta g$$

② Momentum (Exponentially Decaying Average of Past Gradients)

- Since SGD suffers from slow convergence and oscillates in steep regions back-and-forth, spending all its energy moving L ↔ R instead of moving down the valley.

- Momentum solves this by adding memory of past gradients
- Velocity is the memory vector that remembers which way we were already moving

$$v \leftarrow \alpha v - \eta g$$

- Here, α → momentum / friction ($\alpha = 0.9$ means keep 90% of the speed I had in the previous step)
  g → current gradient

$$\theta \leftarrow \theta + v$$

- Since we are averaging the gradients across the steep walls, opposite directions cancel each other out, less zig-zag. Also, velocity builds up over time.

Eg:- Only recent gradients matter, if $\alpha = 0.9$
Previous step's influence = 0.9
2 steps ago = $(0.9)^2 = 0.81$
3 steps ago = $(0.9)^3 = 0.729$

- Nesterov Momentum uses the idea of looking ahead first, then computing gradient, which reduces overshooting and faster + smoother convergence

$$\theta' = \theta + \alpha v$$
$$v \leftarrow \alpha v - \eta \nabla L(\theta + \alpha v)$$
$$\theta \leftarrow \theta + v$$

Note: Key Comparison: SGD = walking downhill step by step
Momentum = rolling ball downhill
Nesterov = looking ahead before stepping

→ So far, we have only tweaked the gradients, but we only updated weights with fixed learning rate for all parameters. But in reality, some features are sparse (updated rarely, slow learning) while some are dense (dominate learning, frequently updated).

- Earlier, Delta-Bar-Delta algo adjusted learning rate based on gradient sign
  Same sign repeatedly → $\eta \uparrow$
  Sign changes → $\eta \downarrow$
  However this works only in full batch and not practical

(3) AdaGrad

- Each parameter gets its own learning rate.
- Keep running sum of all past squared gradients

$$r \leftarrow r + g \odot g$$

$$\theta \leftarrow \theta - \frac{\eta}{\delta + \sqrt{r}} \odot g$$

($\odot$ → element-wise; δ = small constant to avoid zero-division)

- Large past gradients → $\eta \downarrow$
  Small gradients → $\eta \uparrow$

Pros: Great for sparse data, automatically scales updates
Cons: r keeps increasing forever, η becomes too small training can stop early.

④ RMS Prop

- To fix the accumulation of r, use moving average instead of full sum.
- ρ controls decay rate of past gradients (0.9)

$$r = \rho r + (1-\rho) g^2$$
$$\theta = \theta - \frac{\eta}{\sqrt{r} + \delta} \odot g$$

- Keeps recent gradients and forgets old gradients, more stable
- Prevents learning rate from vanishing to zero.

⑤ Adam

- Combines the best properties of Momentum and RMSProp
  (speed) (scaling)
- First Moment (Mean), $\hat{v} = \rho_1 v + (1-\rho_1) g$
  Second Moment (Variance), $\hat{r} = \rho_2 r + (1-\rho_2) g^2$
  Since initialized at zero at $t=0$, bias corrected using.

$$\hat{v} = \frac{v}{1-\rho_1^t}, \quad \hat{r} = \frac{r}{1-\rho_2^t}$$

Weight Update, $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{r}_t} + \delta} (\hat{v}_t)$

- Moves in direction of average gradient ($\hat{v}$) but scale the step size based on how much gradient usually vibrates ($\sqrt{\hat{r}}$)

☆ Second-Order Methods

- Use both first derivative (gradient) and second derivative (Hessian) to optimize

- Based on Taylor Expansion, Newton's method approximates function using:

$$J(\theta) \approx J(\theta_0) + \nabla J(\theta_0)^T(\theta - \theta_0) + \frac{1}{2}(\theta - \theta_0)^T H(\theta - \theta_0)$$

(current loss) (gradient) (Hessian / quadratic)

Then minimizes this approximation

Here H → Hessian of J wrt θ evaluated at θ_0

$$\theta \leftarrow \theta_0 - H^{-1}\nabla_\theta J(\theta_0)$$

where $\nabla J(\theta_0)$ → gradient

H^{-1} → inverse of Hessian (matrix of 2nd deriv)

(measures curvature of loss function)

- Hessian is a square matrix of 2nd order partial derivatives. For n parameters, Hessian is $n \times n$ matrix.. $H_{ij} = \frac{\partial^2 J}{\partial \theta_i \partial \theta_j}$

Here diagonal elements tell curvature along single parameter

Non-diagonal elements tell how gradient of one parameter changes when you move another (interaction between)

- If slope is steep → take bigger step
- If slope is flat → take smaller step

Cons:
- Computationally expensive (inversion of $n \times n$ matrix $$$)
- Memory heavy (storing Hessian)
- Not practical for large deep learning model architectures

☆ Regularization Methods

– Dropout & DropConnect

- Dropout ~~drops~~ deactivates neurons randomly to improve generalization. (Activation)

- DropConnect drops individual weights (connections) with the help of a binary mask (random) such that network becomes randomly thinned at connection level (Parameter-level)
  Normal Layer, $z = Wx$
  In DropConnect, $z = (W \odot M)x$ $(M \in \{0,1\})$

- However can lead to slower training and harder optimization compared to Dropout.

– Batch Normalization (reparameterization technique)

- Normalize outputs of a layer for each mini-batch, leading to bell-shaped distribution, centered around mean ≈ 0.
  Variance ≈ 1
  stable training

$$\hat{x} = \frac{x - \mu_{batch}}{\sqrt{\sigma^2_{batch} + \varepsilon}}$$

$$y = \gamma \hat{x} + \beta$$

- Deep networks suffer from internal covariate shift where distribution of inputs to each layer keeps changing.

Pros: Faster training ($\eta \uparrow$), stabilizes network, regularization effect (prevents vanishing/exploding) (less overfitting)

★ Convolution Neural Networks

- Extracts features by applying filters (kernels) over input.

Ex:- Input Image: $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, Kernel: $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

Output(1) = $(1\times1) + (2\times0) + (4\times0) + (5\times-1) = 1-5 = -4$

- Each filter detects a specific pattern (edges, textures, shape)
- Output of convolution = feature map (response of filter at that position) (multiple filters to capture diff features)
- Stride → step-size of filter movement ∝ $\frac{1}{\text{Output Size}}$
  Padding → adds extra border (usually zeros)
  Without it, output shrinks after every layer
  (with padding = 1, input: $3\times3 \rightarrow 5\times5$
- Output = $\frac{N - F + 2P}{S} + 1$
- $O(i,j) = \sum_m \sum_n I(i+m, j+n) \cdot K(m,n)$
- Each output depends on a small region of image, leading to less computation and better efficiency.
- Same kernels/filters reused, fewer parameters, better generalization.

- If input shifts / rotates, output also changes (feature map changes) CNN detects pattern regardless. $f(g(x)) = g(f(x))$

- Pooling → Downsampling operation that summarizes nearby values
[Convolution → Activation (ReLU) → Pooling]

(i) MaxPooling → max (region) : $\begin{bmatrix} 1 & 5 \\ 2 & 3 \end{bmatrix} \rightarrow 5$

(ii) Average Pooling → mean (region) : $\begin{bmatrix} 1 & 5 \\ 2 & 3 \end{bmatrix} \rightarrow 2$

(iii) L2 Pooling → $\sqrt{\Sigma x^2}$

- Pooling reduces dimensions, smaller feature maps, leading to faster computation. Also helps in translation invariance

- CNN Full Pipeline:
Input → Conv → ReLU → Pool → Conv → ReLU → Pool → ....
→ Fully Connected (Hidden Layer) → Output

Pros: • Automatic feature learning, fewer parameters (weight sharing) spatial awareness, translation handling (equivalence + invariance)

- Flatten → Converts multi-dimensional feature map into 1D vector $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \rightarrow \underbrace{[1 \quad 2 \quad 3 \quad 4]}_{(2\times 2\times 1)}$

- Fully Connected Layer → takes the extracted features and maps to output classes for final decision making.

- Softmax $\equiv \frac{e^{z_i}}{\Sigma e^{z_i}}$ (Used for multi-class classification)
(Outputs between 0 and 1 such that sum of all outputs = 1)

— LeNet

- Small foundational model used for digit recognition (MNIST).
- Architecture: Input → 32×32×1
  Conv → 6 filters (5×5) → 28×28×6
  AvgPool → 14×14×6
  Conv → 16 filters → 10×10×16
  AvgPool → 5×5×16
  FC → 120, FC → 84, Output → 10

— AlexNet

- Deeper than LeNet with ReLU activation, Dropout, Data Augmentation, GPU training
- Architecture: Input → 224×224×3
  Conv1 → 96 filters (11×11), stride = 4, padding = 0
  (55×55×96)
  MaxPool1 → (3×3), stride = 2 (27×27×96)
  Conv2 → 256 filters (5×5), stride = 1, padding = 2
  (27×27×256)
  MaxPool2 → (3×3), stride = 2 (13×13×256)
  Conv3 → 384 filters, (3×3), stride = 1, padding = 1
  (13×13×384)
  ~~MaxPool3 → (3×3), stride = 2 (6×6×~~
  Conv4 → 256 filters (3×3), stride = 1, padding = 1
  (13×13×256)
  MaxPool3 → (3×3), stride = 2 (6×6×256)

FC1 → 4096, FC2 → 4096, FC3 → 1000

— VGG-16

- Use many small filters (3×3) instead of large ones

- Architecture: Input → 224 × 224 × 3
  Conv (64, 3×3, S=1, P=1) → (224 × 224 × 64)
  Conv (64, 3×3, S=1, P=1) → (224 × 224 × 64)
  MaxPool (2×2, S=2) → (112 × 112 × 64)
  Conv (128, 3×3, 1, 1) → (112 × 112 × 128)
  Conv (128, 3×3, 1, 1) → (112 × 112 × 128)
  MaxPool (2×2, S=2) → (56 × 56 × 128)
  Conv (256, 3×3, 1, 1) → (56 × 56 × 256)
  Conv (256, 3×3, 1, 1) → (56 × 56 × 256)
  MaxPool (2×2, S=2) → (28 × 28 × 256)
  Conv (512, 3×3, 1, 1) → (28 × 28 × 512)
  Conv (512, 3×3, 1, 1) → (28 × 28 × 512)
  MaxPool (2×2, S=2) → (14 × 14 × 512)
  MaxPool (2×2, S=2) → (7 × 7 × 512)
  Flatten → 25088
  FC1 → 4096, FC2 → 4096, FC3 → 1000

— PlacesNet

- CNN trained for scene recognition, based on AlexNet

Note: GradCAM is a visualization technique showing where CNN is focusing (heatmap or image shows important regions)

★ Long-Short Term Memory (LSTM)

- A special type of RNN designed to learn long-term dependencies, allowing gradients to flow through time without vanishing
- Regular RNNs accumulate information over time, once stored, cannot forget easily. LSTM uses gates to let the network remember useful info and forget irrelevant info.
- Used for speech recognition, machine translation, and handwriting generation.
- **Cell State** (c_t): Long-term memory that flows across time, minimum modification (conveyor belt)
- **Hidden State** (h_t): Short-term memory, output at each step
- **Gates**: Forget gate, Input gate, Output gate. (decide what to forget, keep or update)

① **Forget Gate**: $f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$ (decides what to forget)

② **Input Gate**: $i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$ (decides what to add)

③ **Candidate Memory**: $g_t = \tanh(W_g[h_{t-1}, x_t] + b_g)$ (new information)

⑤ **Output Gate**: $o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$ (decides output)

④ **Update Cell State**: $c_t = f_t \cdot c_{t-1} + i_t \cdot g_t$ (forget old info, add new info)

⑥ **Hidden State**: $h_t = o_t \cdot \tanh(c_t)$

✱ Gated Recurrent Unit (GRU)

- Simpler architecture of LSTM with fewer gates, same goal. (designed to handle sequences and solve vanishing gradient problems)
- In GRU, forget + input → Update Gate
  Cell state + hidden state → single state
- Update Gate (z_t) → decides how much past info to keep
  Reset Gate (r_t) → decides how much past info to forget.

✱ Deep Generative Models (DGM)

- Model that learns how data is generated to create new data, understand structure (learn patterns) and infer missing values (fill gaps in data), using deep neural networks to model complex probability distributions.

— Boltzmann Machines

- Probabilistic neural network that models distributions over binary variables

$$P(z) = \frac{e^{-E(z)}}{Z} \quad (\text{normalization constant})$$

- Lower energy → higher probability.

$$E(z) = \frac{-1}{2} z^T U z - b^T z$$

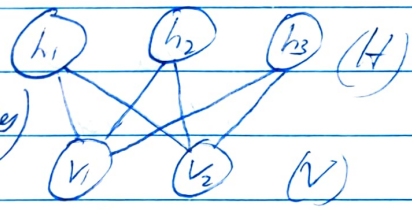
U → weights, b → bias, z → binary variable

- Energy measures how good a configuration is.
- Hidden units capture latent patterns such that BM learns universal approximator of distribution.

— · Restricted Boltzmann Machines (RBM)

- Simple Boltzmann Machine with restricted connections (no intra-layer connections) (generative stochastic NN)
- Two layers; Visible layer (V) containing input data
  Hidden layer (h) learns features

1. Input Data → Visible Layer
2. Compute hidden activations (guess features)
3. Reconstruct input (hidden → visible)
4. Compare with original
5. Update weights (Contrastive Divergence)



(bipartite graph)

- Deep Belief Networks are stacks of RBMs having multiple hidden layers, connected hierarchially.
- Used for feature extraction, dimensionality reduction (PCA), recommendation systems, building blocks of DBNs.

$$E(v,h) = -\sum b_i v_i - \sum c_j h_j - \sum v_i W_{ij} h_j$$

(biases)

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

(hidden given visible)

$$P(h_j = 1 \mid v) = \sigma\left(\sum W_{ij} v_i + c_j\right)$$

(visible given hidden)

$$P(v_i = 1 \mid h) = \sigma\left(\sum W_{ij} h_j + b_i\right)$$