Deep Learning: subset of ML that uses Artificial Neural Networks with many hidden layers to automatically learn patterns and features from data.

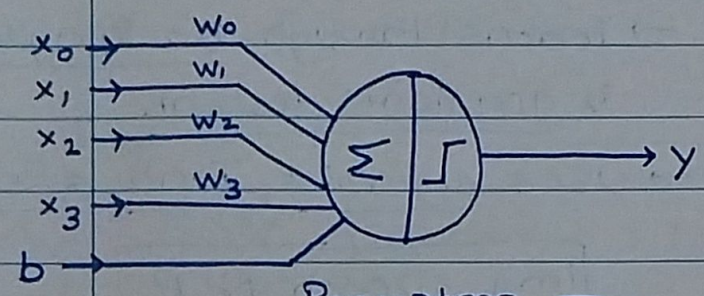
- No manual feature extraction
- for large unstructured data

Artificial Neural Network (ANN)

- computational model inspired from biological neurons.
- a network of nodes (Neurons) has weights, biases, activation functions, inputs & outputs.

✱ Perceptron: {working of Artificial Neuron}

$$y = \alpha\left(\sum_{i=0}^{n} x_i \times w_i + b\right)$$



Perceptron.

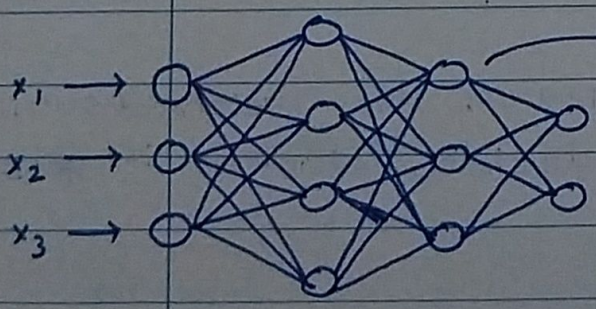
- ⊙ simplest type of artificial neuron.
- ⊙ taken in multiple inputs processes it through weights and biases, adds them up
- ⊙ finally applies an activation func to produce output.

x_i → input
b → bias
w_i → weight
⌐ → activation func.
Σ → sum
y → output

{Single-layer Perceptron}

⊙ Multilayer Perceptron (MLP): type of an ANN consisting of input, output and one or more hidden layers.

→ each neuron of each layer works like a perceptron.



each one is a perceptron

↳ can be drawn for DL architecture

input layer — hidden layers — output layer

[in an MLP, all previous layer perceptron outputs are the inputs of the next layer.]

↳ This is what a Fully connected FNN is.

Advantages:

1) solves non-linear problems
2) understands complex patterns
3) used in both regression and classification.

Feed forward Neural Network (FNN)

- → simplest form of ANN.
- → Information flows in one direction only.
- → No feedbacks or loops exist in the network.
- → MLP is a type of FNN.

types of FNNs

Single-Layer FNN

- → single-layer perceptron
- → input + output layer only
- → simple and low computation
- → solves only problems which are linearly-separable
- → uses perceptron learning rule.
- → uses step function as activation function

application: Logic Gates

Multi-Layer FNN

- → multi-layer perceptron
- → input + hidden + output layer
- → complex & high computation.
- → solves both linear & non-linear problems.
- → learns through backpropagation & gradient descent.
- → uses sigmoid, Tanh, ReLU

Img, speech, NLP

input (x) → numerical values given to perceptron. each represents a feature.

output (y) → final result after applying activation to sum of weighted inputs.
→ usually 0 or 1

weights (w) → parameters that decide the importance of each input.
→ each x_i is multiplied by a w_i

bias (b, w_0) → a constant added to the weighted input's sum
→ helps improve learning by shifting activation function.

Activation Function → a mathematical function applied on the weighted sum of inputs of a neuron that decides whether it gets activated.

- $y = 0$ → not activated
- $y = 1$ → activated.

- ⊙ introduces non-linearity to the network
- ⊙ helps NN learn complex relations.
- ⊙ w/o this network would abe a simple linear model.
- ⊙ e.g. Step, Sigmoid, Tanh, ReLu, Softmax

1) Step Function $f(x) = \begin{cases} 0 & x<0 \\ 1 & x \geq 0 \end{cases}$ Range: $\{0,1\}$

2) Sigmoid Function $f(x) = \frac{1}{1+e^{-x}}$ Range: $(0,1)$

3) Softmax Function $f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$ Range: $(0,1)$

$\sum_{i=1}^{n} f(x_i) = 1$

4) Tanh Function $f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ Range: $(-1,1)$

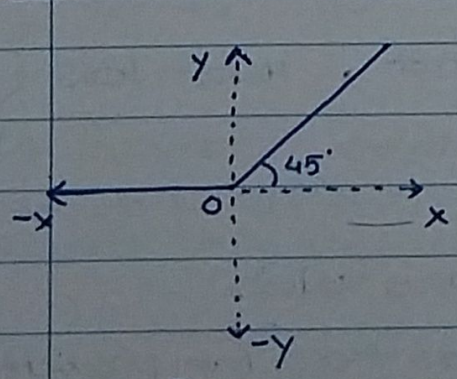
---

5) ⊙ Rectified ~~Unit~~ Linear Unit (ReLU) $f(x) = \max(0,x)$ Range: $[0,\infty)$

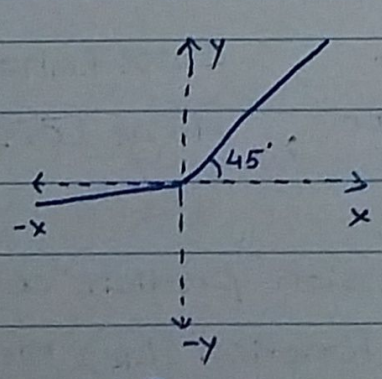
6) ⊙ Leaky ReLU (LReLU) $f(x) = \begin{cases} x & x>0 \\ \alpha x & x \leq 0 \end{cases}$ Range: $(-\infty,\infty)$

↓ very small positive constant

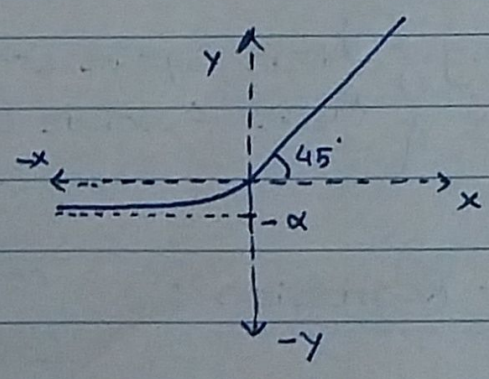
7) ⊙ Exponential RELU (EReLU) $f(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x<0 \end{cases}$ Range: $(-\alpha,+\infty)$



ReLU



LReLU



EReLU

ReLU	LReLU	EReLU
→ causes vanishing gradient problem (For -ve inputs only)	→ reduces it	→ reduces it further
→ low computation	→ slightly higher	→ higher.
→ used as a default choice for most DL models. ★	→ used when ReLU shows many dead/non-activated neurons. ★	→ used when computational cost is acceptable and smoother convergence is needed. ★

While explaining MLP, draw and label each weight, input and output → can also write formulae as e.g. ⊛

* **Back Propagation** : a supervised learning algorithm used to train NN.
  → goal is to minimize errors by updating w_i & b using gradient of loss function.

Steps:

1) Forward Pass → inputs are passed through the network and outputs are calculated.
2) Error calculation → outputs are compared with target/expected outputs to calculate errors.
3) Backward Pass → error is passed on to previous layers and weights are updated using rules of calculus.
4) Weight update →

$$w_{new} = w_{old} - \eta \frac{\partial L}{\partial w}$$

$\frac{\partial L}{\partial w}$ → partial derivative of loss func. wrt weight

η → learning rate

loss function — mathematical func measuring diff. b/w actual and predicted outputs of a neural network.

e.g. { Mean Square Error , Cross-Entropy Error , Huber Loss
Mean Absolute Error , Hinge Loss }

For Regression : regression predicts continuous values
Loss function (L_R) measures how far predicted value is from actual value.

e.g. MSE :

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

n → total no. of outputs
y_i → i^{th} actual output
$\hat{y}_i$ → i^{th} predicted output

MAE:

$$L_R = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|$$

For Classification : classification predicts discrete class labels
Loss function (L_C) measures how well predicted probabilities match actual classes.

e.g. Cross-Entropy error (Log Error) :

$$L_c = \frac{-1}{n}\sum_{i=1}^{n}\left[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)\right]$$

(Binary classification)

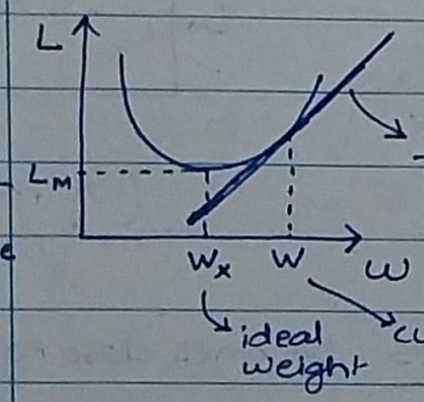
OR

$$L_c = -\sum_{i=1}^{n} y_i \log \hat{y}_i$$

Gradient Descent : optimization algorithm used to minimize the loss function in NN.

(gradient based learning) It updates weights/bias in opp direction of the loss gradient.

e.g.



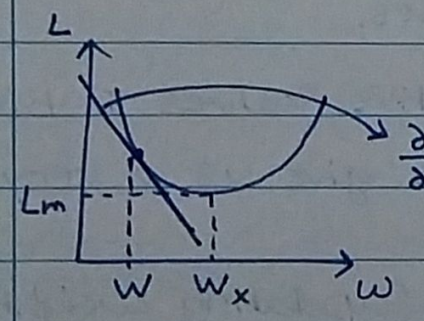
$\frac{\partial L}{\partial w}$ → it is positive here → means w has to be updated in negative direction

minimum loss possible

ideal weight

current weight

decreasing value by $\eta \frac{\partial w}{\partial L}$ we can move closer to w_x



$\frac{\partial L}{\partial w}$ → it is negative here → means w has to be updated in positive direction

increasing value by $\eta \frac{\partial w}{\partial L}$ we can move closer to w_x

(*) This front-back movement of w goes on until is reaches very close to w_x

$$\therefore w_{New} = w_{OLD} - \frac{\partial L}{\partial w} \times \eta$$

(η → learning rate.)

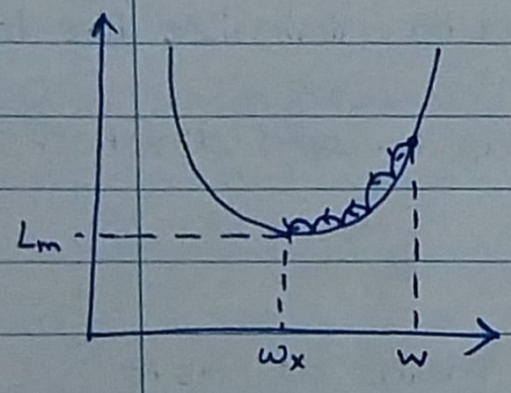
Gradient Descent Formula

Advantages: simple & effective
works for regression as well as classification

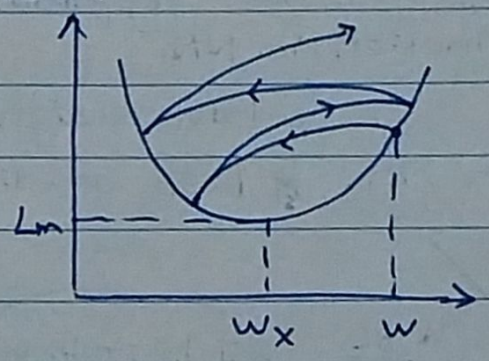
Limitations : can get stuck at local minima
sensitive to choice of η.

* **learning rate** (η) : it is a hyper parameter in gradient descent that controls the step size taken while updating weights.

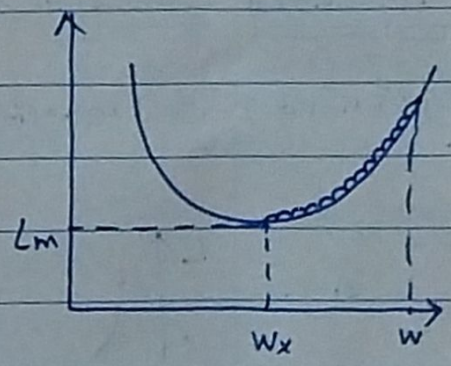
* **Significance** :
  1) η too small → training becomes slow
  2) η too big → algo can fail to converge
  3) η must be appropriate valued to have convergence and that too fast.



(η just right)



(η too big)



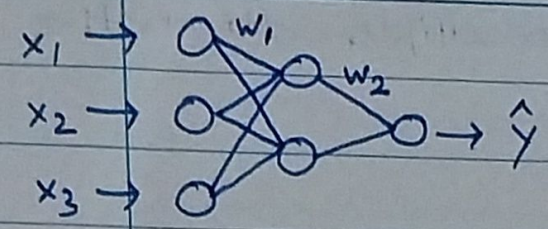
(η too small)

* **Vanishing Gradient** : gradient ($\frac{\partial L}{\partial w}$) vanishes (becomes close to 0) after few back passes.

→ for certain activation functions, the derivative values lie b/w 0 & 1
→ in such cases multiple back passes can give us a very small value of $\partial L/\partial w$
→ which means after few layers, learning (updating weights) either becomes slow or stops altogether.

Result → training becomes inefficient
→ model accuracy reduces
→ very severe problem in deep NN.

Solutions:
1) use better activation functions
ReLU, LReLU, EReLU which avoid shrinking gradients
2) use weight initialization techniques (Xavier init., He Init.) to keep variance of activations stable across layers.
3) use batch normalization which normalizes inputs to each layer, keeping values in a manageable range & reducing vanishing radients
4) Gradient clipping : restricting gradients to not go below certain threshold.



$$\left\{ w_{1\,new} = w_{1\,old} - \eta \frac{\partial L}{\partial w_1} \right\}$$

by chain rule of calculus $\rightarrow$ $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial w_2} \times \frac{\partial w_2}{\partial w_1}$

image a deep NN with multiple such layers : $\boxed{\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial w_n} \times \frac{\partial w_n}{\partial w_{n-1}} \times \dots \frac{\partial w_2}{\partial w_1}}$

this is what slow/No learning is

(*) Each of these gradients < 1

i.e. $\frac{\partial L}{\partial w_1}$ will be closer to 0 so much that $\boxed{w_{1\,new} \approx w_{1\,old}}$

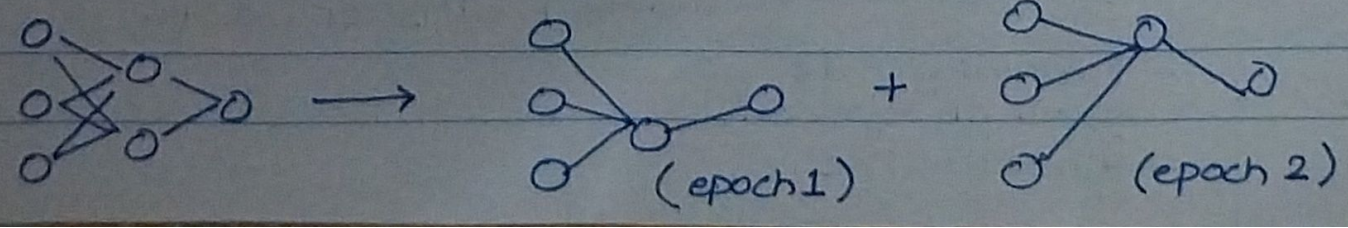
{ (⊙) since ReLU, EReLU, LReLU are producing values in range of ∞, gradients > 1 in most cases which will solve this problem }

(*) **Regularization** : set of techniques used in NN to reduce overfitting by adding constraints to the learning process.

→ prevents model from becoming too complex and memorizing training data, thus improving generalization.

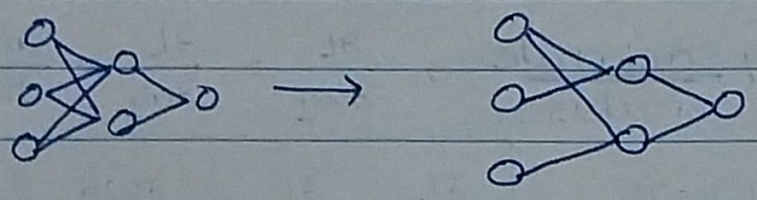
need
1) Deep networks are prone to ovefitting due to millions of parameters
2) without this model may perform well on training but bad on testing data.
3) It ensures that model learns essential patterns only. & not noise

Types : 1) **Dropout** : randomly drop neurons during training
→ this forces NN to not depend on specific neurons and improves generalization.



2) Dropconnect : similar to dropout. But here, it randomly drops weights, not neurons.

→ this forces network to rely on multiple alternative connections.



3) L1 regularization : adds the sum of absolute weights to the loss function

(Lasso)

$$L = Loss + \lambda \Sigma |w_i|$$

↓

Parameter.

→ This encourages sparsity (many weights → 0)

Check Notes

4) L2 regularization : adds the sum of squared weights to the loss function

$$L = Loss + \lambda \Sigma (w_i)^2$$

→ This prevents very large weights & stabilizes training

(*) Hyper Parameters : external parameters set before training a NN.

→ not learned from data.
→ manually set by designers
→ to control learning process & model behavior

Types :

1) model-related : these define architecture of the network

→ no. of hidden layers
→ no. of neurons
→ the activation function used.

2) training-related : these control how the model learns and trains on data.

→ learning rate
→ no. of epochs
→ batch size

3> regularization-related : thes help prevent overfitting and improve generalization.

→ dropout rate
→ weight decay (λ)
→ Early stopping patience.

no. of neurons/layer ↑ more training time
high capacity
overfitting

↓ low capacity
underfitting
No complex data captured.

no. of layers ↑ capture complex patterns
hard to train
more data req.

↓ only simple problems
limited ability to learn.

Optimization algorithms ↪ techniques to minimize loss function by updating parameters (W; & b) during training.

⊛ core of gradient descent / variants

→ Faster convergence
→ avoid local minima
→ good with large data
→ generalization
} need