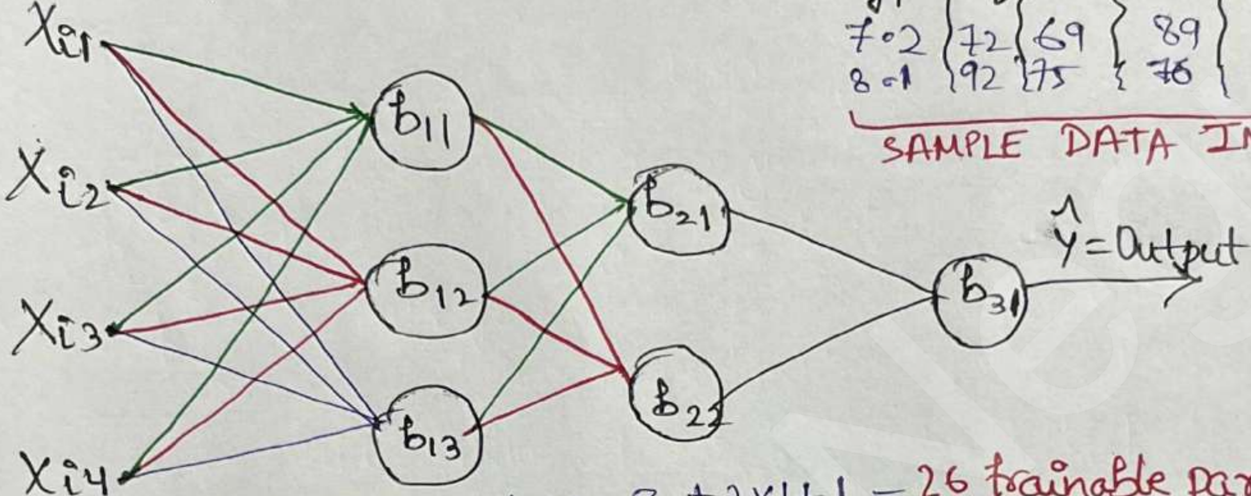
Forward Propogation

→ In simple terms it is the process where the input data travels through the neural network (from left to right) to produce an output.

→ Think of it as testing or prediction Phase.

cgpa	iq	10th	12th	placed
7.2	72	69	89	1
8.1	92	75	78	0

SAMPLE DATA INPUT



$4\times3+3=15 \quad +3\times2+2=8 \quad +2\times1+1 = 26$ trainable parameters.

Prediction → $\sigma(W^T X + b)$ [σ = Activation f^n; W^T = transpose of weights, b = bias]

- Basically output of every perceptron is passed through an Activation f^n 'σ'.

Layer 1:

{Each row - Weights leaving a input}
[Each column - Weights entering next node]

$$W = \begin{bmatrix} w_{11}^1 & w_{12}^1 & w_{13}^1 \\ w_{21}^1 & w_{22}^1 & w_{23}^1 \\ w_{31}^1 & w_{32}^1 & w_{33}^1 \\ w_{41}^1 & w_{42}^1 & w_{43}^1 \end{bmatrix}_{4\times3} \longrightarrow W^T = \begin{bmatrix} w_{11}^1 & w_{21}^1 & w_{31}^1 & w_{41}^1 \\ w_{12}^1 & w_{22}^1 & w_{32}^1 & w_{42}^1 \\ w_{13}^1 & w_{23}^1 & w_{33}^1 & w_{43}^1 \end{bmatrix}_{3\times4}$$

Now Acc. to prediction formula:-

$$\underbrace{W^T \cdot \begin{bmatrix} X_{i1} \\ X_{i2} \\ X_{i3} \\ X_{i4} \end{bmatrix}}_{\underbrace{3\times4 \times 4\times1}_{3\times1}} + \underbrace{\begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \end{bmatrix}}_{+\,3\times1} = \xrightarrow[\text{sigmoid}]{\text{After applying}} \begin{bmatrix} O_{11} \\ O_{12} \\ O_{13} \end{bmatrix}$$

Layer 1 Output
↓
This is input for next layer

Layer 2

$$\begin{bmatrix} w_{11}^2 & w_{12}^2 \\ w_{21}^2 & w_{22}^2 \\ w_{31}^2 & w_{32}^2 \end{bmatrix}^T \begin{bmatrix} O_{11} \\ O_{12} \\ O_{13} \end{bmatrix} + \begin{bmatrix} b_{21} \\ b_{22} \end{bmatrix} \xrightarrow{\sigma} \begin{bmatrix} O_{21} \\ O_{22} \end{bmatrix}$$ Layer 2 Output

$(3\times 2)^T \rightarrow 2\times 3$ $\quad 3\times 1 \quad$ 2×1

Similarly for Layer 3, which gives us final Prediction output.

Loss functions in Deep Learning

★ Loss f^n – Calculated for a single data point. It tells how far off the prediction was for one specific row/example.

★ Cost f^n – The average (or sum) of all those individual losses across the entire dataset. Gives single number representing models total error.

★ Gradient & 'hill' – Think of cost f^n as a landscape of hills & valleys. The gradient is like a compass that points which way is 'uphill'. Since we want to minimize error, we move in opp direction of the gradient (downhill)

Weights are like a knob we use to move our position on that hill.

→ learning Rate – Is the step size taken in that direction.

Epochs – the number of times we repeat this whole cycle (1000)

Loss function in Deep learning

- Object Detection
  - Focal loss
- Regression
  - MSE
  - MAE
  - Huber loss
- Classification
  - Binary Cross Entropy
  - Categorical Cross Entropy
  - Hinge loss
- Autoencoders
  - KL Divergence
- GAN
  - Discriminator loss
  - Min-max gan loss
- Embedings
  - Tripled loss

Mean Squared Error (MSE) / Squared Loss / L2 Loss

Loss $f^n = (y_i - \hat{y}_i)^2$

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

Output Layer must have a Linear Activation f^n

ADVANTAGES

- Easy to interpret
- Differentiable
- Have one local minima

DISADVANTAGES

- Error has diff unit than input
- Not Robust to Outliers

Mean Absolute Error (MAE) L1 Loss

Loss $f^n = |y_i - \hat{y}_i|$

Cost $f^n = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|^2$

Same condition as MSE, as regression & we trying to predict continuous values not within any specified range.

ADVANTAGES

- Intuitive & easy to understand error.
- Same unit as input
- Robust to outliers

DISADVANTAGES

- Not Differentiable.

Huber Loss

- Combines benefits of MSE(L2) & MAE(L1) into single f^n.
- Behaves like MSE when error is small & MAE when error is large (outliers)

$$\text{Loss } f^n = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & \text{for } |y-\hat{y}| \le \delta \\ \delta|y-\hat{y}| - \frac{1}{2}\delta^2 & \end{cases}$$

δ threshold when to act as MSE or MAE

$$\text{Cost } f^n = \begin{cases} \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}(y-\hat{y})^2 & \text{for } |y-\hat{y}| \le \delta \\ \frac{1}{n}\sum_{i=1}^{n} \delta\left(|y-\hat{y}| - \frac{1}{2}\delta\right) & |y-\hat{y}| > \delta \end{cases}$$

Binary Cross Entropy (log loss)

- Used for Binary Classification

Loss f^n:-

$-y\log(\hat{y}) - (1-y)\log(1-\hat{y})$

Cost f^n:-

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

Output Layer must have sigmoid Activation f^n.

- This has multiple local minimas.

Categorical Cross Entropy (Used in Softmax Regression)

- Used when 3 or more classes.
- labels are one hot encoded

Loss $f^n = -\sum_{i=1}^{K} y_i \log(\hat{y}_i)$, $K=3$ (No. of class)

Cost $f^n = \frac{1}{n}(\text{loss } f^n)$.

Each output node must have softmax Activation f^n

Sparse Cross Entropy

- Variation of Categorical Cross Entropy.
- Doesn't expect labels to be OHE like CCE.

Backpropogation [The What?]

→ It is an algorithm used to train neural network by updating weights to minimise loss.

→ It uses chain rule of calculus to compute gradients.

→ STEPS:-

1) Initialize random weights & bias.

2) Select a point/row.

3) Predict (LPA) → forward propogation using dot product.

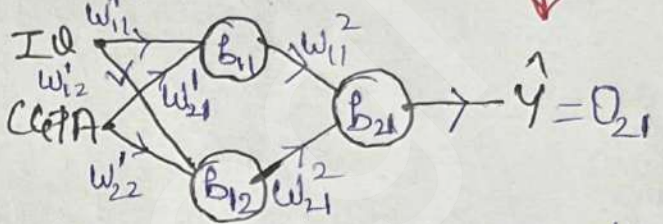
4) Chose a loss f^n (MSE)

5) Update weights & bias using gradient descent $\left[W_{new} = W_{old} - \eta \frac{dL}{dW_{old}} \quad b_{new} = b_{old} - \eta \frac{dL}{db_{old}} \right]$

Sample data

IQ	CGPA	LPA
80	8	3
60	9	5
70	5	8

Each datapoint is sent to this



For $\hat{Y}$(output) $= O_{21}$ we need to update following 'w' & 'b':-

$[b_{21}, w_{21}^2, w_{22}^2, O_{11}, O_{12}]$

O_{11} updation depends on:- $[b_{11}, IQ, CGPA, w_{11}^1, w_{21}^1]$

updation of O_{12} depends on:- $[b_{12}, IQ, CGPA, w_{12}^1, w_{22}^1]$

→ To calculate current we need to calculate previous.

This is Backpropogation of error.

Note - What is derivative = eg $\left\{ \frac{dy}{dx} \right\}$, It means by changing 'x' what is the effect on 'Y', which is SLOPE ★★

To calculate $\left[\frac{dL}{dw_{11}^2} \right] \rightarrow \frac{\partial L}{d\hat{Y}} \times \frac{d\hat{Y}}{dw_{11}^2}$] - Chain rule of differentiation ★★

So we calculate derivatives for each trainable parameter & update them.

→ We repeat step 1-5 in loop for X times, X = no. of data points.

→ This entire process is done epochs = 1000 iterations till cost ≈ 0 or convergence.

So, we can say it is an algorithm that updates neural network weights by propogating error backward using the chain rule to minimize loss.

Gradient Descent in Neural Networks

→ Gradient means 'slope' = most popular algo to optimize N.N.
→ We move opp. direction of slope until we find a valley (minima)
→ Each step-size in that direction is controlled by learning rate (η)
→ So it iteratively adjusts weights & bias in the opposite direction of gradient to find the min of the Loss f^n.

Types Based on data used to compute gradient :-

1) Batch Gradient Descent

- Entire data used in 1 computation
- (Fast - Can jump out of local minima)

Loss f^n traversal

Suppose 50 data points:-

No. of epochs = No. of updates (w, b)
eg epochs = 10, so 10 times we update weights & bias for all 50 data points (full dataset)

Freq of weight update is less

It uses vectorization Technique, dot product {means replacing loops with dot product matrix}

2) Stochastic Gradient Descent

- One sample in 1 computation
- (Noisy/erratic, never fully settles at the bottom)

Loss f^n traversal

Suppose 50 data points:-

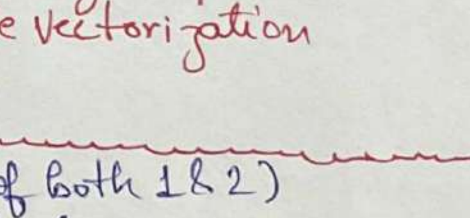
No. of epochs eg = 10, so run a loop for 50 data points and
shuffle → random point → y
w, b update ← loss ←

Frequency of weight update is more

Does not use vectorization

3) Mini Batch Gradient Descent (Best of both 1 & 2)

- Middle ground b/w Batch G.D & Stochastic G.D.
- We make small batches, suppose 10 batches.
- In every epoch we update weights & bias 10 times.
- We use Vectorization here also.
- The loss f^n curve is less spiky than SGD
- We use data batches so no RAM/memory challenges also.



Speed = bgd > Mbgd > SGd

Convergence = bgd < mbgd < sgd

MLP Memoization

→ Memoization means storing results of expensive f^n calls & reusing them when the same inputs come agin, instead of recomputing.

→ It is a time-space trade-off - we use extra memory to get faster computation.

→ In MLP we repeat many similar forward & backward passes & we cache intermediate results like activation or f^n outputs.

Vanishing Gradient Problem in ANN

→ In DEEP neural networks [more layers] when we use Backpropog & sigmoid/tanh Activation f^n then we encounter this problem.

eg - $0.1 \times 0.1 \times 0.1 \times 0.1$ = a very small number.
subtracting this number to update the weight will keep the weights unchanged.

$$\rightarrow w = w - \eta \frac{dL}{dw}$$

Almost 0

Thus, the layers will stop learning

This occurs as in deep Neural Network gradients are computed using the chain rule & are applied (multiplied) many times across layer.

→ How to handle vanishing gradient problem:-

1) Reduce Model Complexity
[Reduce layers/inputs]

2) Using ReLU Activation f^n
[Does not squish input to fixed range]

3) Proper Weight Initialization
[Using Glorot, Xavier]

4) Using Batch Normalization
[Normalize data to a range]

5) Residual Network
[skips connections/provide shortcuts to reach the layers early without being multiplied many times.

IT'S NEVER TOO LATE

-TARUN NEGI

→ Upcoming Notes [Part - 3/3]
Neural Networks Optimization & Performance Tuning.