How to improve the performance of Neural Networks (15)

- Vanishing Gradients
  - Activation f^n
  - Weight Initialization
- Normalization
  - Normalizing inputs
  - Batch Normalization
  - Normalizing Activations
- Optimisers
  - Momentum
  - Adagrad
  - RMS prop
  - Adam
- Hyperparameter Tuning
  - No. of hidden layers
  - Nodes/layer
  - Batch size
- Overfitting
  - Reduce Complexity, ↑data
  - Dropout Layers
  - Regularization (L_1 & L_2)
  - Early stopping
- Gradient Checking & Clipping
- Learning rate Scheduling

1.) Vanishing Gradients

We improve the performance of Neural Networks by solving the vanishing gradient problem that occur during backprop. in deep neural networks using the below methods:-

★ Activation functions

Activation f^n adds non-linearity which help model learn complex non-linear patterns.
Without Activation f^n, a neural network would behave just like a simple linear eq^n no matter how many layers you add.

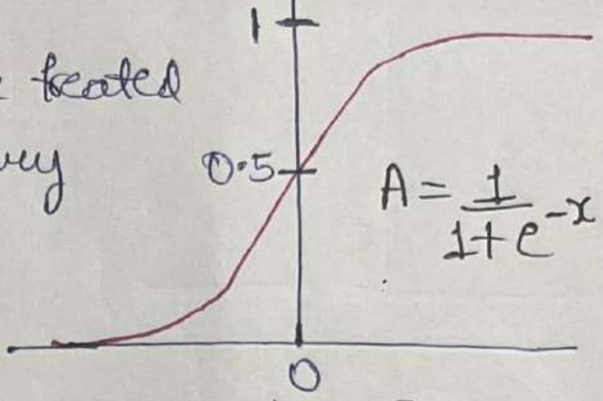
→ What is an Ideal / Good Activation f^n.

- The activation f^n should be non-linear, eg sigmoid $= \frac{1}{1+e^{-z}}$
- It should be differentiable.
- It should be computationally inexpensive.
- It should be zero centered / Normalized.
- It should be non saturating :-
  like, Relu does not squeeze the input b/w a fixed range like sigmoid / tanh squeeze & Vanishing Gradient Occurs.

→ Sigmoid Activation f^n

→ **Advantages:**

- Output is in range [0, 1], can be treated & used in output layer like binary classification as probability
- Can capture Non-linear patterns
- Differentiable f^n.



$A = \frac{1}{1+e^{-x}}$

Range = [0, 1]

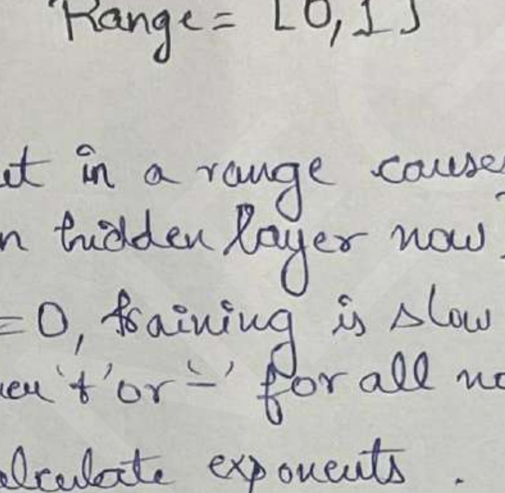
→ **Disadvantages**

- Saturating f^n, Squeezes the input in a range causes Vanishing gradient problem. [Rarely used in hidden layer now].
- Non-Zero centered, mean is not = 0, training is slow because the gradient of one layer is either '+' or '−' for all nodes.
- Computationally expensive as we calculate exponents.

→ Tanh Activation f^n

→ **Advantages:**

- Non-linear
- Differentiable
- Zero centered, both $\left\langle \begin{matrix} +ve \\ -ve \end{matrix} \right.$ gradients thus training is faster.



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

Range = [−1, 1]

→ **Disadvantages:**

- Saturating f^n, prone to Vanishing Gradient
- This is also computationally expensive.

→ ReLU Activation f^n [One of the best Activation f^n today]

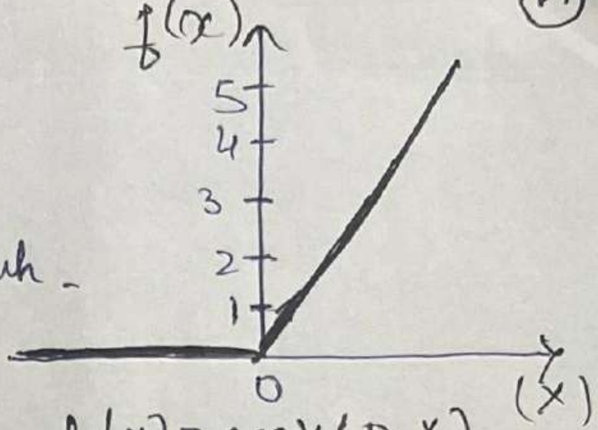
Rectified → Linear → Unit

Rectified ↓ Negative values are set to 0

Linear ↓ Positive values pass as it is

Unit ↓ Single Independent neuron f^n

ReLU only activates positive signals & blocks negative one



→ **Advantages:-**

- Non-linear
- ~~Not~~ Non-saturated in the positive region
- Computationally inexpensive
- Convergence is faster than sigmoid & tanh.

→ **Disadvantages:-**

- Not completely differentiable at 0.
- Not 0 centered {Batch Norm. helps solve this}

$f(x) = \max(0, x)$
range $= [0, \infty)$

→ **Dying Relu Problem:-**

- Relu is $= \max(0, x)$, so for all negative value ReLU outputs 0
So the problem is for negative input gradient $= 0$, weight stops updating so it never learns again.

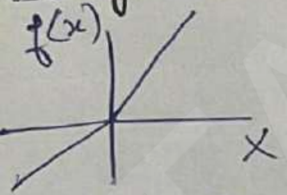
Reasons		Solutions
• Large learning rate.	because of these the value of fn become '−' so the output is 0.	• Set lower learning rate
• Bad weight Initialization		• Use positive weights.
• Bias become '−'.		• Use bias +ve, 0.01 - scientific pro
• Negative Input data		• Use variations of ReLU.

Based on Linear & Non-linear transformations on ReLU

Linear → Leaky Relu, Parametric Relu

Non-linear → ELU, SeLU

Leaky Relu

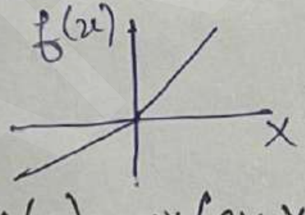


$f(z) = \max\left(\frac{z}{100}, z\right)$

$$\boxed{\begin{cases} z \geq 0 \rightarrow z \\ z < 0 \rightarrow \frac{z}{100} \end{cases}}$$

- For − value output is not 0 it is 0.01z
- Close to 0 centered
- Solves Leaky ReLU Problem

Parametric Relu

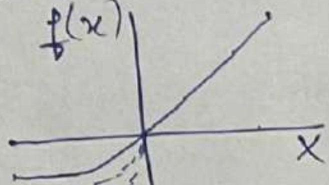


$f(z) = \max(ax, x)$

$$\boxed{f(x) = \begin{cases} x & \text{if } x > 0 \\ ax & \text{otherwise} \end{cases}}$$

- 'a' is a trainable parameter calculated during training
- More Flexible.
- Rest all same as leaky

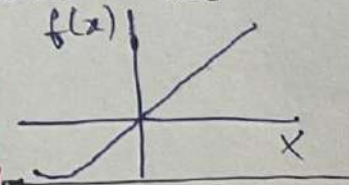
ELU
Exponential Linear Unit



$$\boxed{f(x) = \begin{cases} x & \text{if } x > 0 \\ \underbrace{\alpha}_{\text{constant}}(e^{x} - 1) & \text{if } x < 0 \end{cases}}$$

- This fn is always continuous & always differentiable
- Faster convergence
- Computationally expensive
- Rest all same as ReLU.

SeLU
Scaled Exponential Linear Unit.



$$\boxed{f(x) = \lambda \begin{cases} x & , x > 0 \\ \alpha(e^{x} - 1) & , x \leq 0 \end{cases}}$$

$\boxed{\alpha = 1.67, \lambda = 1.05}$ fixed constants

- Self-Normalizing without using batch Normalization
- More Research is being done on SeLU.

★ Weight Initialization:

→ So step 1 is crucial to initialise the right value to 'w' & 'b' else we might encounter the below problems :-

- Vanishing Gradient Problem.
- Exploding Gradient Problem.
- Slow Convergence.

Steps in Deep learning

1) Initialize the params
2) Chose an optimizing algo (eg GD)
3) Repeat these steps:
- Forward propogate input.
- Compute the cost f^n.
- Compute gradients of cost wrt params using backprop.
- Update each param using gradient

→ What shouldn't be done?
(Wrong ways to initialise weights)

- Zero initialization = $\boxed{w=0,\ b=0}$, No training happens as weights & bias never update with this formula $\left[w = w - \eta \frac{dL}{dw}\right.$
- Non-zero constant = $\boxed{w=1.5,\ b=1.5}$, All nodes behave identical as same activation, behaves like linear model only.
- Random with small weights = $\boxed{\text{np.random.randn(shape)} \times 0.01}$, cause vanishing gradient problem & slow convergence.
- Random with large weights = $\boxed{\text{np.random.randn(shape)}}$, slow training & vanishing Grad., & saturation happens.

The learning is we have to initialise randomly but range matters.

→ What can be done?
(right ways to initialise weights)
Right way is to multiply Rand weights with $\frac{1}{n}$, n = no. of input to node

Methods

Xavier/Glorot (used with tanh)

- Normal
  $= \sqrt{\frac{1}{n}}$
  n = no. of inputs coming to the node
- Uniform
  $= [-\text{limit}, \text{limit}]$
  $\text{limit} = \sqrt{\frac{6}{m+n}}$
  m = input to node
  n = no. of output to node

He init (used with ReLU)

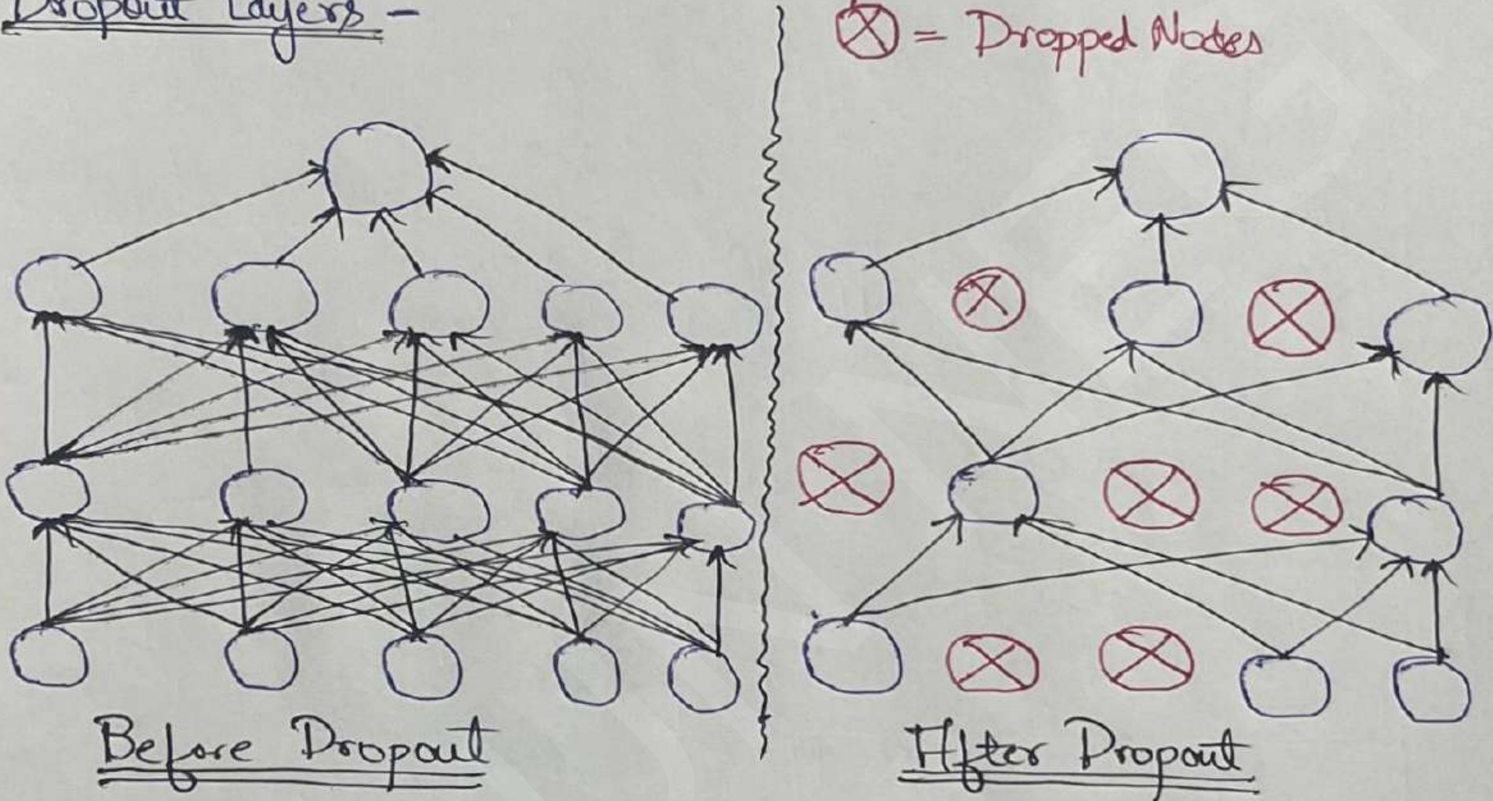
- Normal
  $= \sqrt{\frac{2}{n}}$
  n = no. of inputs to the node
- Uniform
  $= [-\text{limit}, \text{limit}]$ range
  $\text{limit} = \sqrt{\frac{6}{n}}$
  n = no. of inputs to the node

2) Overfitting

→ Neural networks are prone to overfitting as they are complex.
→ Overfitting happens when NN perform well on training & poor on test.
Various ways to handle it are as follows:-

* Reduce Complexity & ↑ data - Basically by reducing no. of nodes or layers & by increasing the input data.

* Dropout Layers -



→ Basically in each epoch we randomly drop ~~nodes~~ nodes based on a factor p that we initialize.
p = 0.5 means randomly drop 50% nodes in each epochs.

→ Dropout matches the random forest analogy because in R.F also we introduce randomness & create multiple model variation & reduce overfitting through ensemble effect.

→ How does prediction work?
- For training we don't have many 'w' randomly - since dropout is ON
- During testing / inferencing dropout is OFF because we don't need randomness during prediction.
- So if more nodes active during ~~training~~ testing, output will be larger so we do weight scaling. $\boxed{\text{If } p = 0.3, \text{ then } w = w(1-p)}$ during testing.

→ Note :-

- Overfitting = Increase value of P.
  Underfitting = Decrease the value of P.
- Start once by applying dropout only on last layer.
- General good range of dropout = 10% to 50%

→ Drawbacks of dropouts :-

- Convergence is slow.
- As loss fⁿ value changes a lot, difficult to interpret gradients/model.

★ <u>Regularization</u> -

- Regularization adds a penalty term to the loss fⁿ, the penalty term punishes the complex models [Large weights should be peanalised]
- ~~Large~~ so that the model does not overfit.

Types

<u>L_2 Regularization (most common)</u>

- If weight is small - penalty is small
- If weight is large - penalty is large

Penalty term: $\lambda \sum w^2$

$Loss = L(y, \hat{y}) + \lambda \sum w^2$

λ controls the strength

Weights are pulled toward zero But never zero.

<u>L_1 Regularization</u>

- Same weight & penalty ratio as L_1
- But penalty ↑ or ↓ linearly

Penalty term: $\lambda \sum |w|$

Loss = Loss + λ. Penalty term

Weights directly pushed to zero
Many weights become exactly zero.

★ <u>Early Stopping</u>.

- Early stopping helps us stop training before overfitting starts.
- We split the data into train & validate data & train epoch by epoch. After each step we check validation loss. If validation loss is <u>improved - continue training</u> else <u>stop training at that epoch</u> if validation loss does not improve for some time.
- Early stopping stops the model where it performs best on unseen data, not where training accuracy is maximum.

3) Normalization

→ Problem without Normalization :-

- Eg Age → 0 to 1000, Salary → 10,000 to 10,00,000, Rating → 1 to 5
- Large - value features dominated & small value features got ignored
- Loss surface becomes zig-zag & learning became slow & unstable.

★ Normalising inputs:

- Normalising makes all inputs comparable & in same scale, centered. So model treats all input features fairly.

Two methods

Standardisation

- Centeres data around mean & scales to standard deviation

$$\boxed{= \frac{X_i - \mu}{\sigma}, \ \mu = \text{mean}, \ \sigma = \text{std.dev}}$$

Normalisation

- Scales data to a fixed range (usually 0 to 1)

$$\boxed{= \frac{X_i - X_{min}}{X_{max} - X_{min}}}$$

★ Batch Normalisation : [Normalization done using batch of data]

- It is used to normalize the output of each layer during training
- It makes training faster, more stable & less sensitive to 'w' initialization
- Output of layer 1 ⟶ input to layer 2, so even if input is standardized internal values keep changing & we apply Batch Normalisation.
  ↳ Internal Covariate Shift - Distribution of inputs to each layer keep changing as weights update.
- So, standardization or Normalization only happen once on input data. But Batch Normalization is applied at every layer [every hidden]

Steps - Neuron give output = $z = wx + b$ → Normalize Activation $(z - \mu / \sigma)$ ↓

Scale & shift $(= \gamma \times \text{Norm.value} + \beta)$ ← Scale = how wide the values should be; Shift = Where center should be

↳ And then Activation f^n is applied

This is done as at this step also all neuron will have same value out so we introduce γ - scale, β - shift.

- During Training - Batch norm. normalizes Activations using mini batch μ & σ ★★★
- During Testing - To normalize use running μ & σ learned during training

Each node is normalized independently but using values from all samples in the batch ★★

4) Gradient Checking & Clipping

These are also techniques used to improve performance & stability of neural networks during training.

→ Gradient Checking = It is a debugging technique used to verify whether backpropogation gradients are computed correctly or not.

$\boxed{\text{If backprop grad} \neq \text{Manually calculated numeric gradient} \rightarrow \text{means bug in code}}$

→ Gradient Clipping = It is used to limit the magnitude of gradients to prevent exploding gradients during training.

$\boxed{\text{If gradient becomes very large} \rightarrow \text{clip it to a fixed limit}}$.

5) Optimisers

→ We use optimizers to speed up the training process.

→ We studied gradient descent & its variation optimizer algos that updates model weights to reduce loss during training but there are few challenges:-

- Single learning rate for every weight / both directions.
- Stuck in local minima possibility.
- Saddle point handling - where slope is zero, so no weight update

→ Exponential weighted Moving Average (EWMA):-
A technique that computes a weighted average where recent ** values get higher weightage & older value decay exponentially.

$\boxed{V_t = \beta V_{t-1} + (1-\beta)\theta_t}$ → In normal average all past values get equal weight in real life recent loss/gradient is more important.

Types of Optimizers

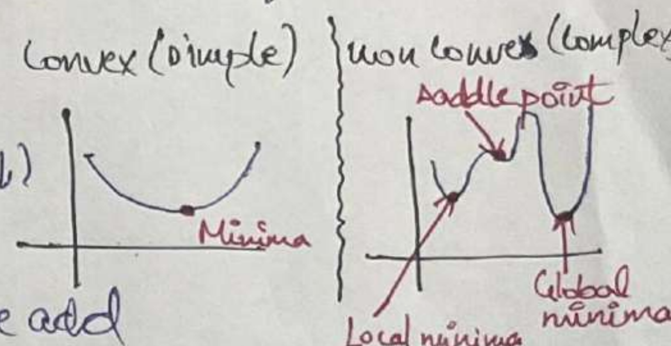
$$\begin{tikzcd}
 & & \text{Types of Optimizers} \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
\text{SGD with momentum} & \text{Nesterov Accelerated Gradient (NAG)} & \text{AdaGrad} & \text{RMS prop} & \text{Adam Optimizer.}
\end{tikzcd}$$

☆ SGD with momentum

→ In complex deep learning the loss f^n is also complex so to reach minima is tough because:-

1) Local minima
2) Saddle point (one side gradient ↑ & other direc ↓)
3) High Curvature - very sharp bend/curve.

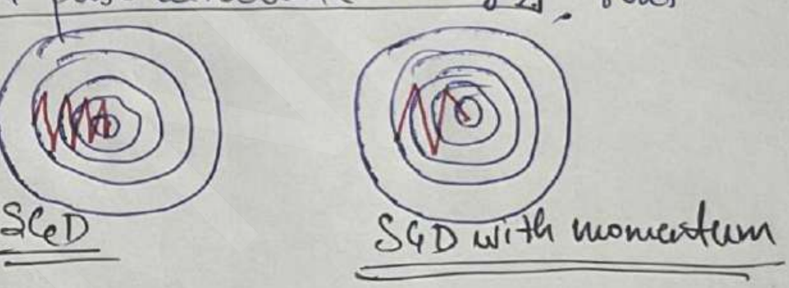
→ So SGD deals with above problems so we add momentum to it & convergence speed increases.



Physics - If a ball is moving downhill, it doesn't stop immediately, it keeps moving because of momentum.

So instead of using current gradient it also remembers past gradients.

$\boxed{\text{Update} = \text{Current gradient} + \text{past direction(memory)}}$. this reduces zig zag and move faster in correct direction & smooth learning



☆ Nesterov Accelerated Gradient (NAG)

→ Problem with SGD with momentum was it could overshoot oscillation

→ NAG solves this By { let me first see where my speed will take me then I'll correct before actually moving

NAG calculates the gradient after looking ahead (peeking) in the momentum direction, not at the current position.

☆ AdaGrad (Adaptive Gradient)

→ Main idea = Different parameters should have diff learning rates.

→ This help in handling sparsity by giving rare features higher learning rate, so sparse features learn quickly & converge fast.

→ Disadvantage :- { Now, we don't use AdaGrad in Deep learning }

- learning rates keep decreasing continuously because it accumulates all past gradients, which can make learning rate extremely small & cause training to stop early.

★ RMS Prop (Root Mean Square Prop)

→ RMSProp improves AdaGrad by using an exponential moving average of squared gradients instead of accumulating all past gradients, thus prevents learning rate from becoming too small.

AdaGrad remembers everything, RMS prop remembers only recent history.

★ Adam Optimizer (Adaptive Moment Estimation) ★★

→ Currently most powerful optimizer, used in ANN, CNN, RNN.
→ Mix of Momentum & RMS prop.

$\boxed{\text{Update = Direction from momentum + Learning rate control from RMSProp}}$

→ Default parameter in Deep Learning.

6) Learning Rate Scheduling

→ Learning rate decides how big step your model will take while learning/training.

Large LR → fast but unstable

Small LR → stable but very slow.

→ LR scheduling idea = Start with higher learning rate & gradually reduce it during training.

→ Common types (step decay, Reduce on Plateau, Cosine Annealing)

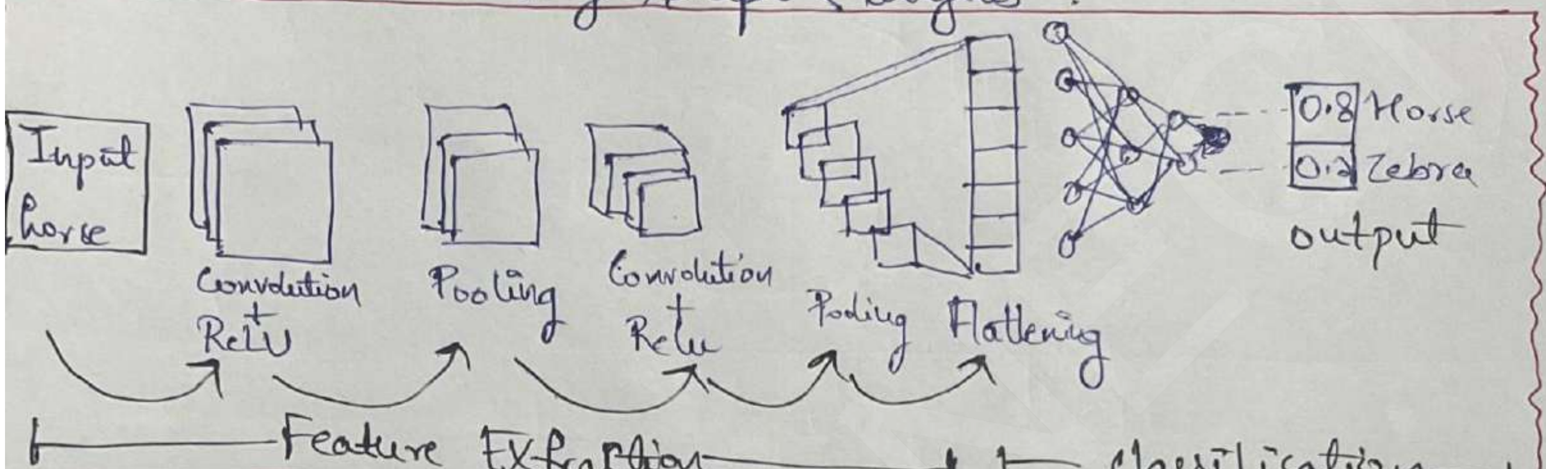
IT'S NEVER TOO LATE

-TARUN NEGI

BONUS

Convolutional Neural Network (CNN)

→ CNN is a deep learning model mainly used for images
→ Images have many pixels & too much data, normal N.N treat images like normal numbers & don't understand shape but CNN understands edges, shapes & weights.



★ Filters/Kernel - Small matrices that slide over input image matrix

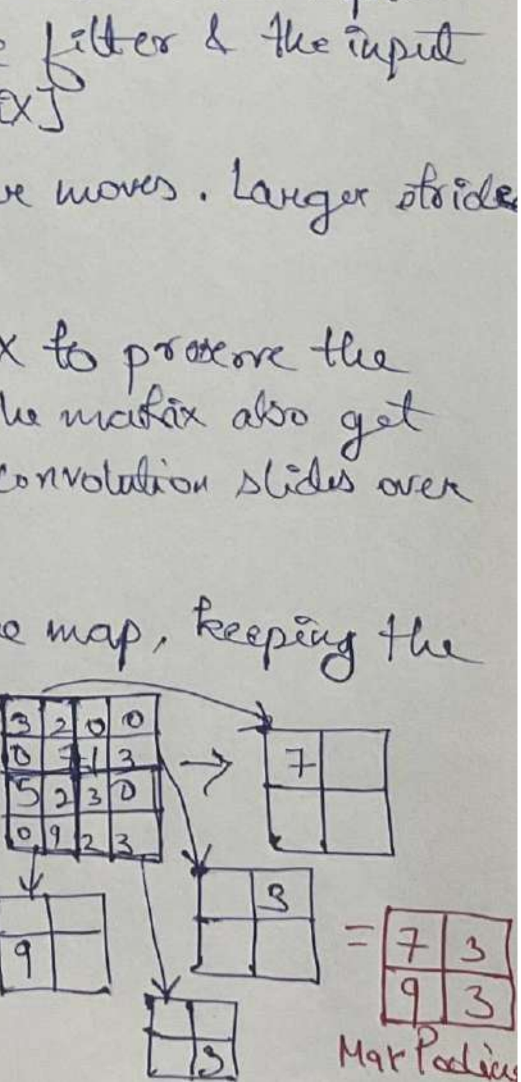
★ Convolution operation - Each filter convules (slides) over the input image, computing the dot product b/w the filter & the input patch. This generates a feature Map [matrix]

★ Stride = The step size by which the filter moves. Larger strides result in smaller feature maps.

★ Padding = Adding zero around input matrix to preserve the spatial dimension. The edges of the matrix also get equal chance/weightage while the convolution slides over it.

★ Pooling = Reduces the dimensions of feature map, keeping the most important information.

- Maxpooling - Takes max value from each patch of feature map.
- Avg pooling - Takes the avg value from each patch of the feature map.



→ Forward propogation in CNN:- (BONUS)

$$\begin{tikzcd}
\text{Input Image} \arrow[r] & \begin{array}{c}\text{Convolution}\\ \text{(Filter slides over input}\\ \text{extract (edges, shapes)}\end{array} \arrow[r] & \text{Feature Maps} \arrow[r] & \begin{array}{c}\text{ReLU}\\ \text{(Removes negative,}\\ \text{Keep '+' activation)}\end{array} \arrow[d] \\
\begin{array}{c}\text{ANN}\\ \text{(Use learned}\\ \text{features \& predict)}\end{array} \arrow[dr] & \begin{array}{c}\text{Flatten}\\ \text{(convert feature}\\ \text{map to 1D vector)}\end{array} \arrow[l] & \begin{array}{c}\text{Repeat CNN layer}\\ \text{(Add complexity)}\end{array} \arrow[l] & \begin{array}{c}\text{Pooling}\\ \text{(Reduces size}\\ \text{Keeping imp info.)}\end{array} \arrow[l] \\
& \begin{array}{c}\text{Loss Calculation}\\ \text{(Compare Prediction vs}\\ \text{Actuals)}\end{array} & &
\end{tikzcd}$$

→ Backpropogation in CNN:- ***

- Error flows backwards, CNN updates its weights.
- ~~Back~~ Calculate loss/gradient & adjust ANN weights using GD.
- Backprop through the flatten layer. (Just reshaping, no learning here)
- Backprop through pooling & Relu.
- And update filter weights through CNN.

→ Transfer Learning

- Transfer learning means using a pretrained model for a new task.
- Model is already trained on large dataset.
- CNN layers are reused for feature learning.
- New fully connected layers are added for classification.
- Saves time, data & computation, works best when dataset is small.