Deep feed-forward neural n/ws:

→ Also called as feed forward neural n/w or multilayer perceptions

→ It consists of multiple layers of interconnected nodes including i/p layer, hidden layers & output layer (info flows in 1 direc)

→ The info flows through the layers from i/p to o/p without feedback loops

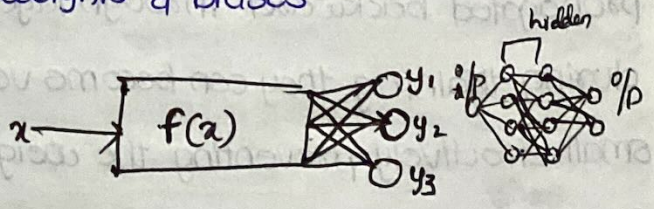
→ They are used for classification & regression

Key components of DFFNNs

1. Neuron (Node): Basic processing unit in a Neural n/w that apply an activation funct^n to the weighted sum of i/ps to produce o/ps
2. Layers:
   - i/p layer: First layer the receive raw data
   - hidden layers: Intermediate layer b/w i/p & o/p layers
   - o/p layer: The final layer produce the o/p of the n/w
3. Weights: parameters that transform i/p data within each layer
4. Biases: Additional parameters that allow activation funct^n to be shifted
5. Activation functions: Non linear functions applied to weighted sum of i/ps at each neuron (ReLU, sigmoid, tanh)

Working:

1. Initialization: Weights & biases are initialized often with small random values
2. Forward propagation:
   - → i/p data is fed into i/p layer
   - → data is passed through hidden layers where each neuron performs a weighted sum of i/ps followed by an activation funct^n
   - → final o/p layer produces the n/ws prediction
3. Loss function: difference b/w n/ws prediction and Actual target value is classified using loss funct^n (Mean Sq^r error for regression, Cross entropy loss for classification)
4. Back propagation: loss is propagated back through the n/w to update weights and biases using GD Alg & other Alg
5. Optimization: Goal is to minimize the loss funct^n by iteratively adjusting the weights & biases



Training:

1. Data preparation: Split data into training, validation and test sets, Normalize or Standardize the data if necessary
2. Model Architecture: Define No. of layers, no. of neurons per layer & Activation funct^{n}s

3. Training: Use a dataset to iteratively update the weights & biases through forward propagation, loss computation, backpropagation & optimization

4. Evaluation: Assess the models performance on validation & test sets to check overfitting or underfitting

5. Hyperparameter tuning: Adjust hyper parameters (learning rate, No. of layers, No. of neurons) to improve model performance

Applications:

→ Image Recognization

→ Natural language processing

→ Regression Analysis

→ classification tasks

Challenges (check)? incomplete

1. Vanishing & Exploiding Gradients:

- Vanishing Gradients: As gradients are propagated backward through layers during training they can become very small, effectively preventing the weights from updating and leading to slow convergence or stalling of training.

- Exploding gradients: Conversly gradients can grow exponentionally as they propagate backward leading numerical instability & divergence of training process

2. Interpretability: Deep learning models are often opaque & difficult to interpret, hard to know why a particular decision was made

3. Explanability: Related to interpretability it is difficult to explain the decision made by DL models

4. Overfitting: The model becomes too specialized to the training data & fails to generalize well to new, unseen data

5. Underfitting: The model is too simple and fails to capture the underlying patterns in the data

6. Data Quality: DL models are only as good as the data they are trained on poor quality data → poor performance

7. Data ~~Imbanede~~ Imbalance: Where one class has a significantly large No. of instances than the others, which can make it challenging to train accurate models.

8. Scalability: Expensive to train (especially for datasets)

9. lack of Domain Knowledge: Which can challenge for those without expertise in the area.

10. Transfer learning: Challenging to adapt pre-trained models to new tasks and domains

learning XOR

perception rule to design XOR

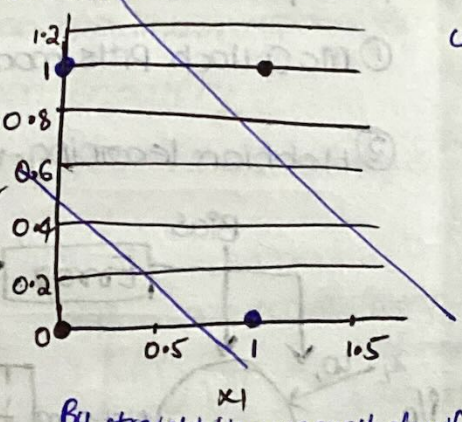
→ single layer perceptron can only solve linearly seperable problems i.e it can find only a straight line to seperate different classess. XOR funct^n is not linearly seperable so no straight line

→ multi layer perceptron (deep NN) can solve XOR by using atleast one hidden layer with non-linear activation funct^{n}s.

Example:

consider the truth table

x_1	x_2	y
0	0	0 ✓
0	1	1 ✗
1	0	1 →
1	1	0 ✓



By straight line we can't classify

→ we use eq^n $y = x_1\bar{x}_2 + \bar{x}_1 x_2$

rewrite as $y = z_1 + z_2$

where

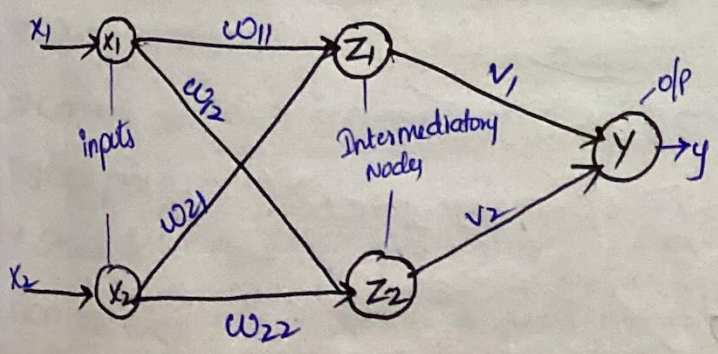
$z_1 = x_1\bar{x}_2$ (funct^n 1)

$z_2 = \bar{x}_1 x_2$ (funct^n 2)

$y = z_1$ (or) z_1 (funct^n 3)

Activation funct^n

$$f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} \geq \theta \\ 0 & y_{in} < \theta \end{cases}$$



First function: $z_1 = x_1\bar{x}_2$ (1 0)

x_1	x_2	z_1
1	0	1
0	1	0
1	1	0

Assume initial weights of $w_{11} = w_{21} = 1$

Threshold = 1 learning rate = 1.15

→ (0,0) ⇒ $z_{1in} = w_{ij} * x_i$

$= w_{11} * x_1 + w_{21} * x_2$

$= 1 * 0 + 1 \times 0 = 0$ (out = 0)

→ (0,1) ⇒ $z_{1in} = w_{ij} \times x_i$

$= w_{11} \times x_1 + w_{21} \times x_2$

$= 1 \times 0 + 1 \times 1 = 1$ (out = 1)

the target output of (0,1) is 0 so expected ≠ target so update the weights

$$w_{ij} = w_{ij} + \eta(t - o)x_i$$

(w_{ij}: old weight; η: learning rate; t: target o/p; o: calculated o/p)

$w_{11} = 1 + 1.5 * (0 - 1) \times 0 = 0$ ($x_1 = 0$)

$w_{21} = 1 + 1.5 \times (0 - 1) \times 1 = -0.5$ ($x_2 = 1$)

$w_{11} = 1 \quad w_{21} = -0.5$

→ (0,0) ⇒ $z_{1in} = w_{ij} \times x_i = 1 \times 0 + (-0.5) * 0 = 0$

(out = 0)

→ (0,1) ⇒ $z_{1in} = w_{ij} \times x_i = 1 \times 0 + (-0.5) \times 1 = -0.5$ ($y_{in} \leq \theta$)

(out = 0)

→ (1,0) ⇒ $z_{1in} = w_{ij} \times x_i = 1 * 1 + (-0.5) * 0 = 1$ ($y_{in} \leq \theta$)

(out = 1)

→ (1,1) ⇒ $z_{in} = w_{ij} \times x_i = 1 \times 1 + (-0.5) \times 1 = 0.5$ ($y_{in} \geq \theta$)

(out = 0)

$Z_2 = \bar{x}_1 x_2$

x_1	x_2	Z_2
0	0	0
0	1	1
1	0	0
1	1	0

Assume initial $w_{12} = w_{22} = 1$

Threshold = 1

learning rate = 1·5

$\rightarrow (0,0) \rightarrow Z_{2in} = w_{ij} \times x_i = 1\times0 + 1\times0 = 0$ (out = 0) ($y_{in} \geq \theta$)

$\rightarrow (0,1) \rightarrow Z_{2in} = w_{ij} \times x_i = 1\times0 + 1\times1 = 1$ (out = 1) ($y_{in} > \theta$)

$\rightarrow (1,0) \rightarrow Z_{2in} = w_{ij} \times x_i = 1\times1 + 1\times0 = 1$ (out = 1)

calculated ≠ target

$-w_{ij} = w_{ij} + \eta(t-o)x_i$

$-w_{12} = 1 + 1·5(0-1)\times1 = -0·5$

$-w_{22} = 1 + 1·5 \times (0-1)\times0 = 1$

$w_{12} = -0·5 \quad w_{22} = 1$

$\rightarrow (0,0) \Rightarrow Z_{2in} = w_{ij} \times x_i = (-0·5)\times0 + 1\times0 = 0$ (out = 0)

$\rightarrow (0,1) \rightarrow Z_{2in} = w_{ij} \times x_i = (-0·5)\times0 + 1\times1 = 1$ (out = 1)

$\rightarrow (1,0) \rightarrow Z_{2in} = w_{ij} \times x_i = (-0·5)\times1 + 1\times0 = -0·5$ (out = 0)

$\rightarrow (1,1) \rightarrow Z_{2in} = w_{ij} \times x_i = (-0·5)\times1 + 1\times1 = -0·5$ (out = 0)

$y = z_1 (\text{or}) z_2 \qquad y_{in} = z_1 v_1 + z_2 v_2$

Assume initial weights $v_1 = v_2 = 1$

Threshold = 1

learning rate = 1·5

x_1	x_2	i/p z_1	i/p z_2	y
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1
1	1	0	0	0

$\rightarrow (0,0) \rightarrow y_{in} = v_i \times z_i = 1\times0 + 1\times0 = 0$ (out = 0)

$\rightarrow (0,1) \rightarrow y_{in} = v_i \times z_i = 1\times0 + 1\times1 = 1$ (out = 1)

$\rightarrow (1,0) \rightarrow y_{in} = v_i \times z_i = 1\times1 + 1\times0 = 1$ (out = 1)

$\rightarrow (0,0) \rightarrow y_{in} = v_i \times z_i = 1\times0 + 1\times0 = 0$ (out = 0)

Theory: Perceptron learning

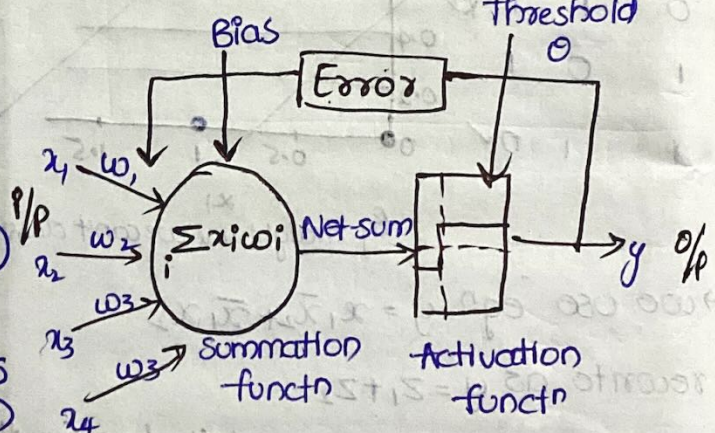
→ first NN model perceptron in 1958

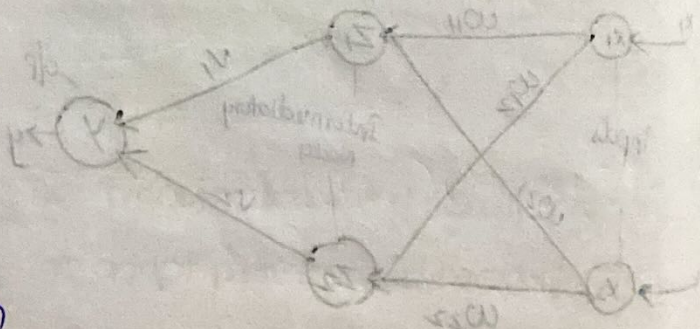
→ perceptron is a linear binary classifier used for supervised learning

→ perceptron is combination of 2 concepts

① McCulloch Pitts model of an artificial neuron

② Hebbian learning – rule for adjusting weights





Gradient Based learning

→ It is a ML approach that relies on optimization algorithms to adjust model parameters

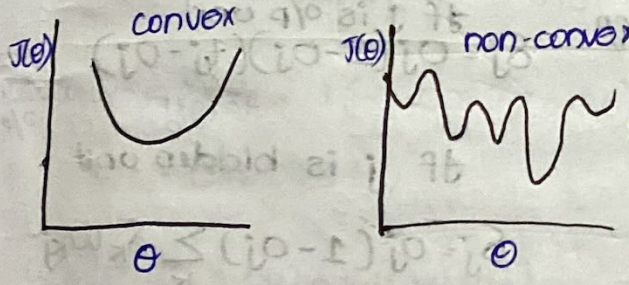
→ This process is typically implemented using backpropagation

→ The gradients are computed recursively through the layers of a neural n/w

→ This process allows models to learn complex patterns and representation from data in learning in neural n/ws

→ Major difference b/w linear models & Neural n/w is non-linearity

→ Nonlinearity causes loss funct^{n}s to become 'nonconvex'



→ NNs are trained using iterative GB optimizers that drive cost funct^n to a very low value, rather than the linear eq^n solvers used to train linear regression models/convex optimization Algs with guaranteed global convergence

→ Convex optimizat^n converges starting from initial parameters

→ Stochastic Gradient descent applied to non-convex loss funct^{n}s doesnot have such convergence guarantee and is sensitive to the values of initial parameters

Hidden units:

→ Refer to the neuron's or nodes with a NNs hidden layer

→ They play a crucial role in learning complex patterns from data

Types

1. Fully connected (Dense) layer:

All nodes are connected to every node in the previous & following layers.

2. Convolutional layers

→ Used for Image-related tasks

→ Neurons are connected in grid to local regions in i/p

3. Recurrent layers

→ Used in Recurrent RNN's for sequential data

→ They have connections that form cycles.

Architecture Design:

Key step/area in building effective deep learning models

→ Refers to the creation and configuration of NN architecture

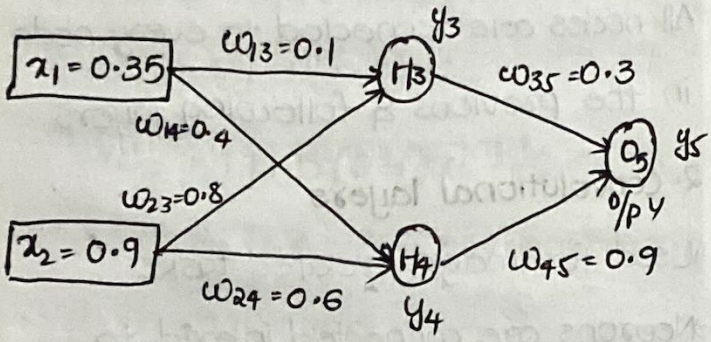
→ This involves deciding the no. of layers & type of layers, The connect^n b/w these layers & the ~~architecture~~ activation funct^{n}s used

→ Common architecture include convolutional NN for image processing, Recurrent NN for sequential data

Back propagation Algorithm

(Back propagation of error)

Ex: Assume neurons have sigmoid activation funct^n, perform a forward pass and a backward pass on the network. Assume that actual o/p of y is 0.5 & learning rate is 1 perform another forward pass



① Forward pass: compute o/p for y_3, y_4 & y_5

$a_j = \sum_j (w_{ij} * x_i)$ $\quad y_j = f(a_j) = \frac{1}{1+e^{-a_j}}$ ← Activation funct^n

$H_3 \Rightarrow a_1 = (w_{13} \times x_1) + (w_{23} \times x_2)$

$= (0.1 \times 0.35) + (0.8 \times 0.9)$

$= 0.755$

$y_3 = f(a_1) = \frac{1}{1+e^{-0.755}} = 0.68$

$H_4 \Rightarrow a_2 = (w_{14} \times x_1) + (w_{24} \times x_2)$

$= (0.4 \times 0.35) + (0.6 \times 0.9)$

$= 0.68$

$y_4 = f(a_2) = \frac{1}{1+e^{-0.68}} = 0.6637$

$y_5 \Rightarrow a_3 = (w_{35} \times y_3) + (w_{45} \times y_4)$

$= (0.3 \times 0.68) + (0.9 \times 0.6637)$

$= 0.801$

$y_5 = f(a_3) = \frac{1}{1+e^{-0.801}} = 0.69$ (N/w o/p)

The Expected o/p = 0.5

Error = $y_{target} - y_5$

$= 0.5 - 0.69$

$= -0.19$

Each weight changed by:

$\Delta w_{ji} = \eta \delta_j o_i$

η = learning rate

δ_j = error at j^{th} unit

o_i = o/p at i^{th} unit

If j is o/p unit

$\delta_j = o_j(1-o_j)(t_j - o_j)$ (target o/p, calculated o/p)

If j is hidden unit

$\delta_j = o_j(1-o_j)\sum_k \delta_k w_{kj}$

$t_j \rightarrow$ correct teacher o/p for unit j

$\delta_j \rightarrow$ Error measure for unit j

② Backward pass: compute δ_3, δ_4 & δ_5

for o/p unit:

$\delta_5 = y(1-y)(y_{target} - y)$

$= 0.69 \times (1-0.69) \times (0.5-0.69)$

$= -0.0406$

For hidden unit:

$\delta_3 = y_3(1-y_3) w_{35} \times \delta_5$

$= 0.68(1-0.68)(0.3 \times -0.0406$

$= -0.00265$

$\delta_4 = y_4(1-y_4) w_{45} \times \delta_5$

$= 0.6637(1-0.6637)(0.9 \times -0.0406)$

$= -0.0082$

Compute new weights:

$\Delta w_{ji} = \eta \delta_j o_i$

$\Delta w_{45} = n \delta_5 y_4 = 1(-0.0406)(0.6637)$

$= 0.0269$

$w_{45}(new) = \Delta w_{45} + w_{45}(old)$

$= -0.0269 + 0.9 = 0.8731$

$\Delta w_{14} = n \delta_4 x_1 = 1(-0.0082)(0.35)$

$= -0.00287$

$w_{14}(new) = \Delta w_{14} + w_{14}(old)$

$= -0.00287 + 0.4 = 0.3971$

Similarly modify all weights

i	j	w_{ij}	δ_j	x_i	η	updated w_{ij}
1	3	0.1	-0.00265	0.35	1	0.0991
2	3	0.8	-0.00265	0.9	1	0.7976
1	4	0.4	-0.0082	0.35	1	0.3971
2	4	0.6	-0.0082	0.9	1	0.5926
3	5	0.3	-0.0406	0.68	1	0.2724
4	5	0.9	-0.0406	0.6637	1	0.8731

③ Forward pass: Compute o/p for y_3, y_4 & y_5

$a_1 = (w_{13} \times x_1) + (w_{23} \times x_2)$

$= (0.0991 \times 0.35) + (0.7976 \times 0.9)$

$= 0.7525$

$y_3 = f(a_1) = \frac{1}{1+e^{-0.7525}} = 0.6797$

$a_2 = (w_{14} \times x_1) + (w_{24} \times x_2)$

$= (0.3971 \times 0.35) + (0.5926 \times 0.9)$

$= 0.6723$

$y_4 = f(a_2) = \frac{1}{1+e^{-0.6723}} = 0.6620$

$a_3 = (w_{35} \times y_3) + (w_{45} \times y_4)$

$= (0.2724 \times 0.6797) + (0.8731 \times 0.6620)$

$= 0.7631$

$y_5 = f(a_3) = \frac{1}{1+e^{-0.7631}} = 0.6820$ (N/w o/p)

→ If we are satisfied with the network o/p it is ok but if we are not satisfied with n/w o/p we repeat the process

Parameter norm penalties $\Omega(\theta)$

This is a technique used to prevent neural n/ws from becoming too complex or overfitting the training data

Parameter: parameters are weights & biases associated with each connection b/w neurons

$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta)$

(Ω - pertinent to standard J; α lies b/w $[0, \infty)$ - hyperparameter; J = objective funct^n)

→ NN aim to learn patterns in data

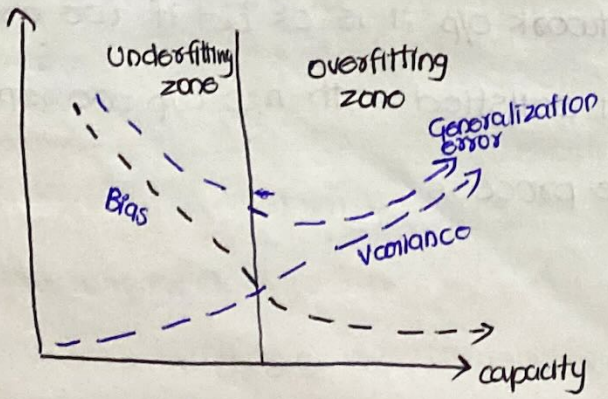
→ Without regularization they might not memorize the training data

There are two types of Regularization techniques

i) L1 Regularization (Lasso)

ii) L2 Regularization (Ridge)

Regularization: It is defined as any modification/change in learning alg that helps reduces its error over a test dataset (generalization error) but not supplied/training data



Optimal capacity

Bias vs variance tradeoff graph here sheds a bit more light on the nuances of Reg & demonstrat^n

i) L1 parameter Regularization

→ It adds the absolute values of to the loss function

[lasso Regression adds absolute value of magnitude of coefficient as penalty to loss function]

→ It shrinks the less imp features coefficient to zero thus removing some features altogether. So this works well for feature selection in case we

LASSO - Least Absolute Shrinkage and Selection Operator

$$\text{loss/cost} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda\sum_{i=1}^{m}|w_i|$$

m = No of features

n = No of examples

y_i = Actual target value

$\hat{y}_i$ = predicted target value

Chars

→ Easy to implement & can be trained as one short thing

→ Robust in dealing with outliers

→ give sparse sol^n

ii) L2 Parameter Regularization

→ Ridge Regression uses L2

→ Ridge regression adds squared magnitude of coefficient as penalty term to loss function.

→ The regularization parameter (λ) penalizes all the parameters except intercept so that the model

generalize the data and doesn't overfit

cons

→ It forces the weights to be small but doesn't make them zero & doesn't give sparse soln

→ Not robust with outliers as square terms blow up the error differences of outliers

→ perform better when all i/p features influence o/p (weights are roughly equal size)

→ It can learn complex data patterns

$$\text{cost} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda\sum_{i=1}^{m} w_i^2$$

m = No of features

n = No of examples

y_i = Actual target value

$\hat{y}_i$ = predicted target value

L1 Regularisation

→ sum of absolute value of weights

→ Sparse soln is outcome

→ Multiple soln's are possible

→ Built in feature selection in penalty term

→ Robust to outliers

→ Used in datasets with sparse features

L2 Regularisation

→ sum of squared weights

→ Non sparse (more segregated) soln

→ One one soln

→ No specific feature selection mechanism

→ Not robust to outliers due to sq^r term

→ Used in dataset with complex features

Elastic Net Regression: It is a combination of L1 & L2 Regularisation

→ That implies we add the absolute norm penalties of weights as well as squared measured of the weights

→ With help of extra hyperparameter(α) that controls ratio of L1 & L2

$$\text{cost/loss} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda\left((1-\alpha)\sum_{i=1}^{m}|w_i| + \alpha\sum_{i=1}^{m} w_i^2\right)$$

Norm penalties as constrained optimization

→ To minimize any function under some constraints we can construct a generalized lagrangian function containing the objective funcⁿ along with penalties

$$L(\theta, \alpha : X, y) = J(\theta; X, y) + \alpha(\Omega(\theta) - k)$$

we get optimal θ by solving langrangian

→ If $\Omega(\theta) > k$ then weights need to be compensated highly, α should be large to reduce its value of below k

→ If $\Omega(\theta) < k$, then norm shouldn't be reduced to much, α should be small.

Penalities	constraints
push weight ⇒ 0	→ define boundry within weights remain

→ large penalities might cause non convex optimization alg to stuck in local minima due to small values of θ leading to formation of dead cells

→ Constraints induce higher stability

Benefits of contraints

→ Avoiding deadcells

→ high stability (high learning rates cause weights to grow too large leading to overflow)

→ Balanced weights

Column norms

→ Reject norm of each neurons weight individually

→ Ensures no single neuron has large weights leading to more balanced & stable n/w

Frobenius norms

→ control overall size of weights of matrix

→ doesn't guarantee that individual weights are small

Under constrained problems

→ This occur when there are more unknowns than eqⁿ leading to multiple possible solⁿs

challenges

- lack of constrainsts result overfitting
- The model might capture noise instead of true patterns

Solⁿ for under constrained problems

→ Regularization Adds constraints indirectly by penalizing certain solⁿs

→ Data Augmentation. ↑ dataset to provide more info

→ Feature selection: Reduce no of features simplifying the model

Data Augmentation:

→ It is a technique to ↑ the diversity of dataset by applying transformations for existing data

→ It enhance model performance by helping it generalize better to neo unseen data

Why to use?

→ DA achieve Invariance (model need to recognize objects regardless of variance) like position, scale etc)

→ Combat overfitting (by ↑ dataset model relies less on specific features)

Types of Augementation

1. Generic transformations: Rotations, flips, translations & scaling. Ensure transformations dont alter class label

2. Color brightness Adjustments: Changes to brightness, contrast, color make model robust to diff lightening conditions

3. Adding Noise: Random noise can make models more resilient to variations

4 Advanced techniques:

cutout: Randomly masking out sections of img

Mixup: Creating new sample by mixing 2 img & their labels

Challenges:

→ label consistency

→ Applicability

Semi supervised learning

→ Combines a small amount of labeled data $P(x, y)$ with large amount of unlabelled data $P(x)$

→ To improve the prediction of $P(y|x)$, the probability of a label given a sample

→ Instead of training 2 models (lab/unlable) we use single model that shares parameters b/w two

like This model has 2 parts

i) Generative model learns the structure of i/p data (x) or joint distribution of i/p & label (x, y)

ii) Discriminative model learns to predict the label (y) from i/p data (x)

→ By sharing parameter b/w models we can leverage the unlabeled data to improve performance Discriminative model

Key concepts

1. Joint probability $P(x, y)$ Represent relnship b/w datapoints (x) & classlabel (y)

2. Marginal distribution $P(x)$ Distribut^n of data point without considering labels

3. Condinal probability $P(y|x)$ This is coant to predict prob of label (y) with data (x)

process

→ Representat^n learning

$h = f(x)$

makes predictions more accurate by ensuring similar i/p leads similar o/p

→ Combining criteria

Supervised learning Uses labled data to directly learn $P(y|x)$

Unsupervised " : Uses unlabeled data to learn structure of $P(x)$, $P(x, y)$

→ shared parameter

Noise Robustness

It involves introducing noise into diff parts of a model to improve its stability & generalization

1. Noise in weights

Adding noise to weights can be seen as a way to handle uncertainty

*Bayesian treats weights as uncertain & model this with a prob distribution

→ Regularization prevent overfitting by stabilizing learning

Ex: Adding Gaussian noise to weights lead to regularization effect

2. Noise in labels (problem) Incorrect labels can mislead the model

Sol^n: label smoothing

Adjust the target probalities to account for label noise

→ for correct label assign probality $1-\varepsilon$

→ for other labels distribute probability $\varepsilon/(k-1)$ k = No of classes

Noise in weights	Noise in labels
Stabilize training by making models less sensitive to small changes	Enhance robustness by reducing reliance on potentially incorrect labels

Multitask learning

It is a technique where a model is trained on multiple related tasks simultaneously

→ Generalization (Goal ↑ how well a model performs on new, unseen data) By using examples for multiple tasks to find patterns that apply broadly not just to specific task

& Types of model parameters

i) Task-specific parameter

→ Unique to each task

→ learn features specific to particular task

ii) Shared parameters

→ Used by multiple tasks

→ Capture common patterns / features across diff tasks

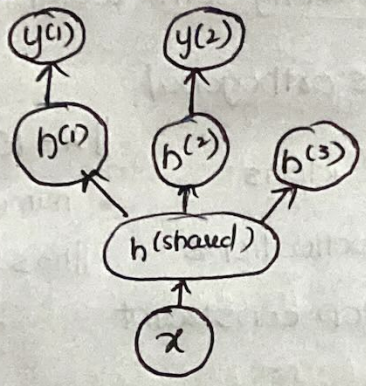
→ It works for shared info, more data, regularization effect

Ex: Task specific — Details of specific shape of cars vs trucks

shared — Common vehicle features like wheels, lights

→ Works best when tasks share some chars

→ Model can learn hierarchical features that capture both specific & general patterns

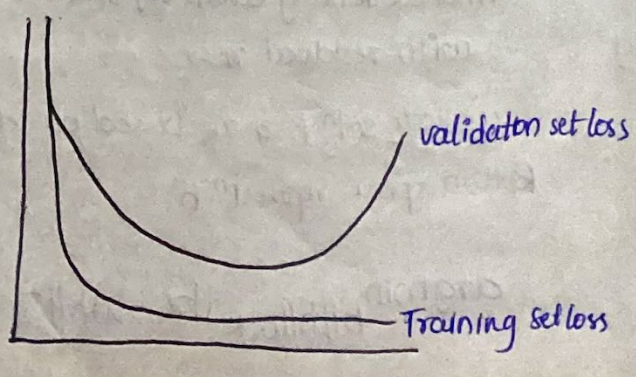


Early stopping

→ It is regularisation technique used to prevent overfitting by monitoring the performance on validatn set

Training error: Error on training data ↓ as model learns

Validation error: Error on unseen validatn data. It ↓ initially but ↑, indicating overfitting



process

i) During training the model learns from training data

ii) Simitaneously it's evaluated on a seperate validatn set

iii) After each epoch calculate the validatn loss

iv) Save the model parameter if the validation loss improves

v) Define a patience parameter (The no of epochs to wait for improvement)

vi) Stop training if no improvement is observed within this period

steps

→ Split data into training & validatn data

→ Initialize model

→ Training loop

- Train for each epoch
- Calculate validatn loss
- save model if validatn loss improves
- check patience & decide to stop if needed

→ Retrain (optional)

Adv:

→ prevents overfitting

→ Reduces computation

→ No extra hyperparameters

Parameter sharing:

→ Instead of learning separate parameters for similar task/i/p you share the same parameters

usecases common arch like CNN, where filters(kernels) are across diff parts of img

Adv:

→ Memory efficient

→ Consistency

Parameter typing:

→ Forces certain parameters to be equal based on prior knowledge about their rel^nships

Ex: if 2 models perform similar tasks you might tie their parameters to encourage learning similar representat^n's

Adv:

→ Improved learning

→ reduced complexity

Sparse Representation

→ It encourage model to activate only a few neurons at a time leading to efficient & interpretable models

Techniques for sparsity

1. Activation penalties

→ penalize non zero activations to encourage sparsity

→ common penalties include L1 Reg^n on activations

2. Orthogonal Matching pursuit (OMP)

→ Aim to find a sparse represent^n by selecting a limited no of non zero entries

→ solves a contrained optmizat^n problem to find sparse rep^n

→ Works effiently with weight matrix W is orthogonal

Implementat^n ideas

$\|x - Wh\|_2^2$ minized with $\|h\|_0 \leq b$

→ penalty on activations

→ Avg activation constraint

Adv:

→ Memory effiency

→ Speed

→ Better Generalizat^n

→ Robust to new data

→ Interpretability

Alg steps OMP

residue $r_0 = x$

for k = 1 to b

find dictionary atom w_j that best correlates with residual r_{k-1}

Update soln h & r_k based on selection

Return spare represt^n h'

$\text{argmin}_h \|h\|_0 \leq k \|x - Wh\|^2$

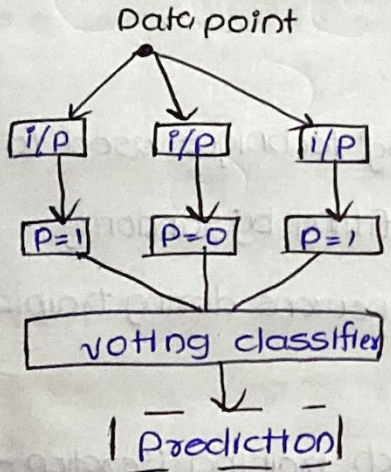
Ensemble learning

→ It is a supervised learning technique used for ~~regression~~ DL to improve overall performance by combining the predictions from multiple models

Methods

i) Voting (Averaging)

Involves making a prediction that is the avg (regression) or the sum (classification) of multiple models



ii) Bagging (Bootstrap Aggregation)

→ It is a meta alg designed to ↑ stability & accuracy of the models like classification & regression

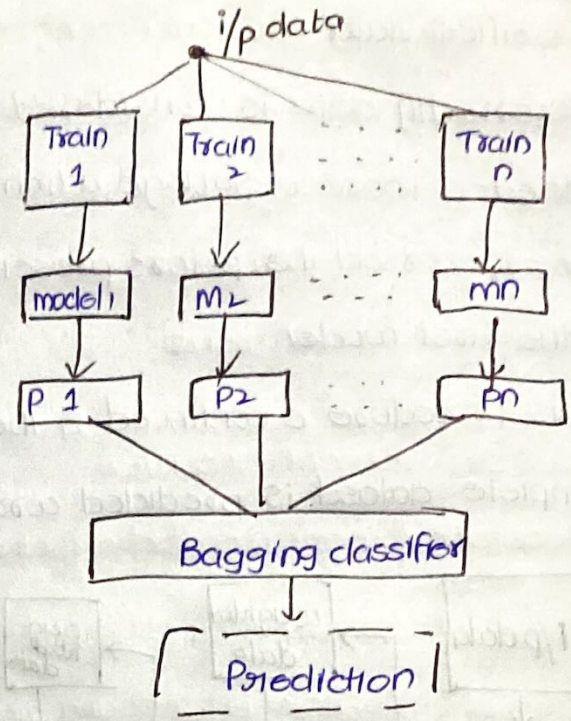
→ It ↓ variance & help to avoid overfitting

→ It is special case of model averaging approach

Adv:

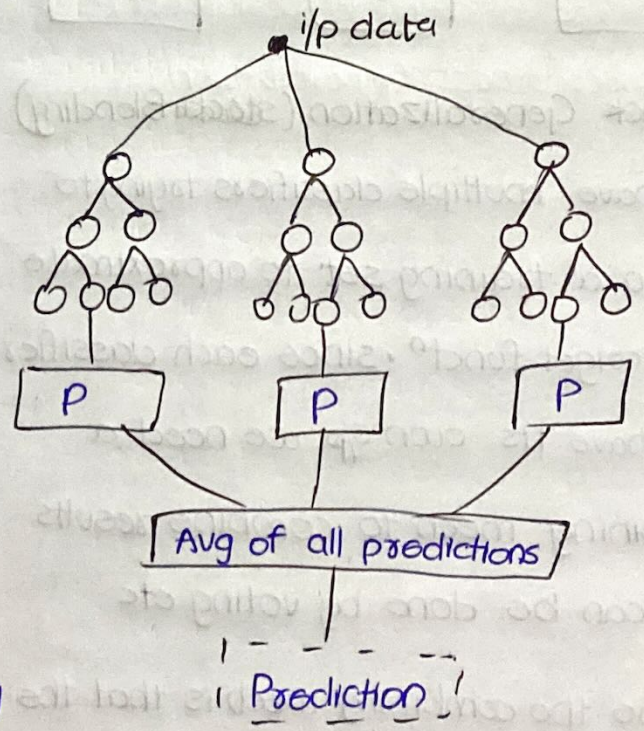
→ Reduces variance

→ Improve stability



iii) Random forest

Multiple DTs are constructed & then they are merged to get more accurate prediction



iv) Boosting:

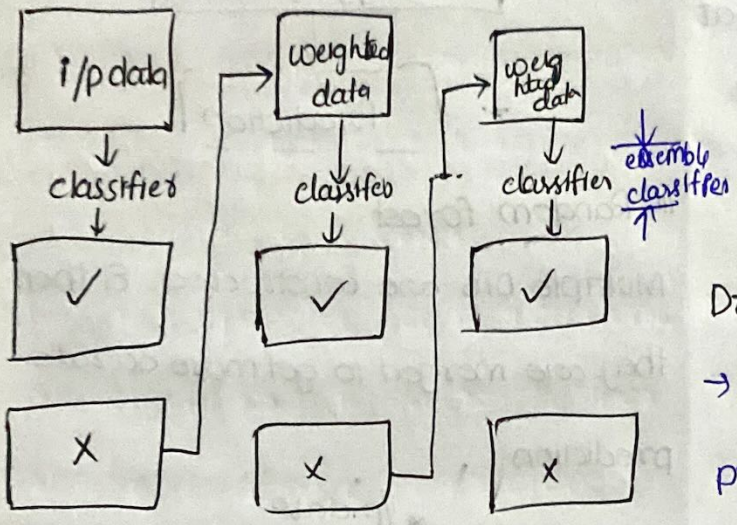
→ It attempts to build a strong classifier from no. of weak classifiers

→ First, a mode is built from training data it has correctly & incorrectly

classified data

→ Incorrectly data is considered & added. a more weightage which tries to correct the errors present in the first model

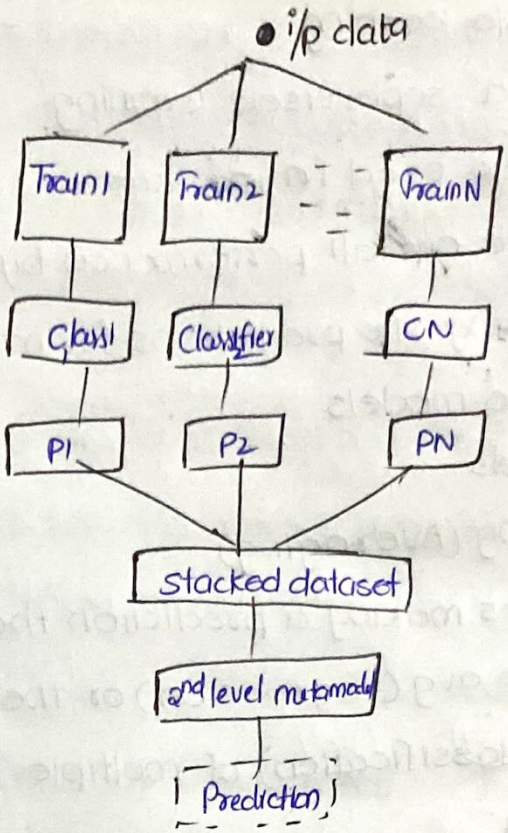
→ This procedure is continued & the complete dataset is predicted correctly



v) Stack Generalization (~~stacked~~ Blending)

We have multiple classifiers trying to fit to a training set to approximate the target funct^n. since each classifier will have its own o/p we need a combining mech to combine results this can be done by voting etc

→ Here the combining mech is that the o/p of classifiers will be used as training data for another classifier to approximate target funct^n



Dropout:

→ It is a Reg^l technique used to prevent overfitting by randomly deactivating neurons during training

Working

→ During each training iteration dropout randomly set of fraction of neurons to zero based on dropout rate p

→ probability p is the hyperparameter determine the proportion of neurons to drop

Common values are:

0.5 ⇒ fully connected layers

0.2 - 0.3 ⇒ convolutional layers

Training phase: Neurons are dropped with probabity p. This forces the n/w to learn redunda

Inference phase: All neurons are active & their o/p are scaled by 1-p to maintain consistent o/p magnitudes

Adv:

→ Reduce overfitting

→ Ensemble effect

Adversal: Slightly altered i/p that cause a modd to make incorrect predictions with high confidence

→ Many dL models are primarily linear so small change in i/p can cause large changes in o/p

Adversarial training

→ ↑ the model robustness by training it on adversarial examples

process

→ Generate adverscarial ex during training

→ Include these ex in the training set

→ Encourage the model to produce the same o/p for original & Adversarial examples

Given model $y = Wx$ a small perturbation ϵ results in We change in o/p

→ Train model to locally constant, ↓ sensitivity to perturbations

Semi supervised learning with AT

→ Unlabeled data: Assign a pseudo-label $\hat{y}(x)$ using the model

→ creat x' such that $y(x') \neq \hat{y}(x)$

→ Ensure x' & x are classified to maintain class consistency on disconnected manifolds

Adv:

→ ↑ Robustness

→ Better Generalization

Target distance

It measures distance blw data points on manifolds by considering small transformations

→ Data is assumed to lie on loco dimensional manifold

→ It approximates manifold near each data point using tangent vectors

Appln

→ finds KNN using this manifold based dist metric

Tangent propagation:

Train models to invariant to small transformations that move data along the manifold

Method:

learning classifier $f(x)$ where the gradient $\nabla f(x)$ is $\perp$ to tangent vectors $V(i)$

Add a penalty term minimizing the directional derivative of $f(x)$ along $V(i)$

→ Focus on resisting small perturbations unlike data augmentation which handles larger transformations

Manifold Tangent classifier

Autoencoder: learn manifold structures through unsupervised learning. Capture the data's manifold by reducing dimensionality & reconstructing i/p & errors

Tangent propagation: Use learned manifolds with tangent prop to ensure classifier invariance to manifold perturbations

Adv: Robustness, Generalization (for all) 3

Chapter - 2

Optimization for Training Deep Models:

- Neural Network training is the most challenging optimization problem in DL, requiring significant computational resources and time. Specialized techniques have been developed to address the issue.
- The Goal is to optimize neural network parameters (θ) to minimize a cost function ($J(\theta)$), which evaluates performance on the training set and may include regularization terms.

How Learning Differs from Pure Optimization

- Training deep models involves optimization algorithms, but there are significant difference between ML optimization & pure optimization.
- In pure optimization, the goal is often to directly minimize a cost function $J(\theta)$, where as in ML, we typically optimize $J(\theta)$ indirectly in the hopes of improving a performance measure (P).

In supervised learning, the cost function often takes the form of an average over training set:

$$J(\theta) = E_{(x,y)\sim \hat{p}_{data}}\left[L(f(x;\theta), y)\right]$$

where L is per-example loss function

$\hat{p}_{data}$ is empirical distribution defined by the training data.

This form is often refered as empirical risk minimization

⇒ Empirical Risk minimization:

Empirical Risk minimization involves minimizing the training set error in hopes that generalization error (or risk) on the true data distribution $P_{data}(x,y)$ will also be minimized.

The empirical risk minimization formula is:

$$\frac{1}{m}\sum_{i=1}^{m} L\left(f(x^{(i)};\theta), y^{(i)}\right)$$

$m \rightarrow$ no. of training examples.

However, this approach can lead to overfitting, especially when models are highly expressive & capable of memorizing training set. & this is unsuitable for gradient-based optimization method. For this reason, empirical risk minimization is often modified by using Surrogate loss functions.

⇒ Surrogate Loss Functions and Early Stopping:

In DL, we often optimize a surrogate loss (eg: negative log-likelihood) instead of the true loss (eg: 0-1 classification loss), as the latter is often intractable to minimize. Early stopping prevents overfitting by halting training when validation performance deteriorates. (degrades)

⇒ Batch and Minibatch Algorithms:

In ML, especially with large datasets, optimization algorithms generally compute updates to the model parameters using a subset of training examples.

Scanned with OKEN Scanner

Instead of calculating gradients over the entire dataset (batch gradient descent), stochastic Gradient descent (SGD) or mini batch gradient descent is used.

- Batch Gradient Descent: Uses the entire dataset for each update.
- Stochastic Gradient Descent (SGD): Updates parameters using one training example at a time.
- Minibatch SGD: Uses small random subsets (minibatches) of data to estimate the gradient.

The Gradient is given by:

$$\nabla_\theta \, \mathbb{J}(\theta) = \mathbb{E}_{x,y \sim \hat{p}_{data}} \nabla_\theta \log p_{model}(x, y; \theta)$$

Several factors influence the choice of minibatch size are:

(1) Larger batches provide more accurate gradient estimates but with diminishing returns.

(2) Smaller batches underutilize hardware like GPUs, which perform better with batch sizes that are powers of two (eg: 32, 64, 128)

(3) Larger batches require more memory, limiting batch size on memory constrained hardware.

(4) Smaller batches introduces noise in gradient estimates, which can help regularize the model and improve generalization.

Scanned with OKEN Scanner

* Challenges in Neural Network Optimization *

Optimization is ~~genod~~ an extremely difficult task. The optimization problem for training neural networks is generally non-convex. Some of the challenges are:

① Ill-conditioning:

The ill-conditioning problem is generally believed to be present in neural network training problems. Ill conditioning can manifest by causing SGD to get "stuck" in the sense that even very small step increase the cost function.

Consider the second order Taylor series expansion of cost function is given by!

$$\frac{1}{2}\epsilon^2 g^T H g - \epsilon g^T g$$

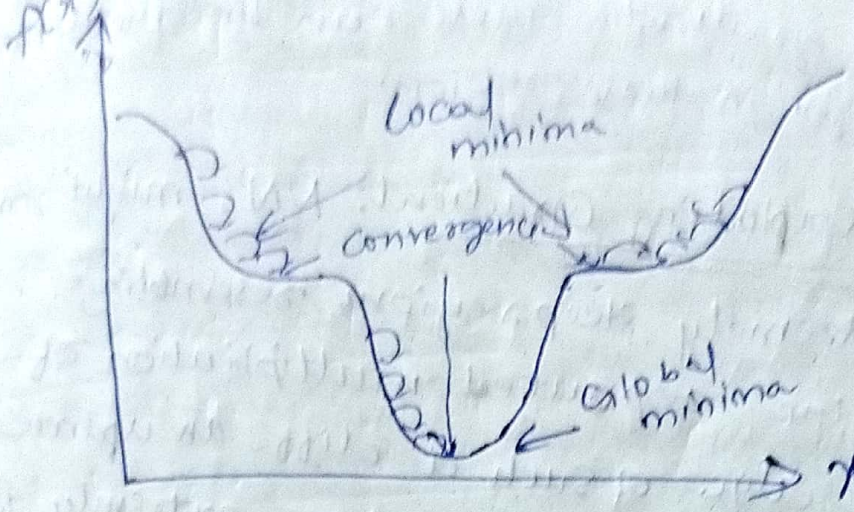
Ill conditioning occurs if $\frac{1}{2}\epsilon^2 g^T H g$ becomes larger than $\epsilon g^T g$ making learning slow despite a strong gradient.

② Local Minima:

Nearly any DL model is guaranteed to have an extremely large number of local minima (LM) arising due to the model identifiable problem.

→ the local minima caused due to the swapping of two hidden nodes leads to same equivalent models. thus not being problem.

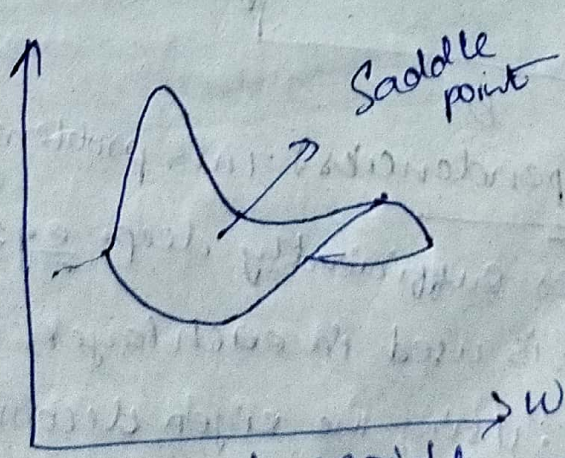
Scanned with OKEN Scanner



→ If local minima with high cost are common, it becomes a serious problem. which leads to the low gradients. Hence, it is common to aim for a low but not minimal cost value.

(3) plateaus, Saddle points and other flat Regions:

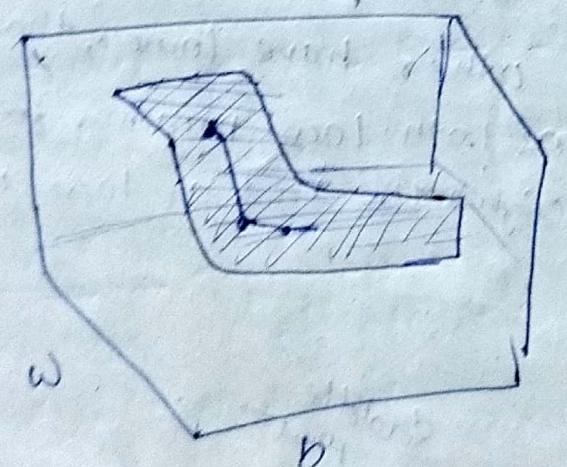
Saddle point (SP) is another type of point with zero gradient where some points around it have higher values and the other have lower. that is, A saddle point acts as both local minima ~~for and~~ ~~also~~ for some neighbours and as local maxima for others.



SGD empirically tends to rapidly avoid encountering a high-cost saddle point. There also might be wide flat regions of constant values thereby

having a zero gradient. These can be problematic if cost is high in these regions.

④ **Cliffs and Exploding Gradient:** NN's might some times have extremely steep regions resembling cliffs due to the repeated multiplication of weights. At the edge of such a cliff an update step might throw the parameters extremly far. It can be taken care by using gradient Clipping. The Gradient indicates only the direction in which to make the update. If GD updates proposes making a very large step the Gradient clipping intervenes to reduce step size.



⑤ **Long term dependencies:** This problem encountered when NN becomes sufficiently deep. eg: If same weight matrix w is used in each layer, after t steps, we would get w^t. Using the eigen decomposition of w.

$$w = V \, diag(\lambda) V^T \qquad w^T = V \, diag(\lambda)^t V^T$$

Thus, any eigen value not near an absolute value of 1, would either explode or vanish leading to vanishing or exploding gradient problem.

Scanned with OKEN Scanner

~~* Optimization Strategies and~~

* Basic Algorithms *

⇒ Stochastic Gradient Descent:

- SGD and its variants are probably most used optimization algorithms in DL.
- SGD is a variant of gradient Descent that works by using a small randomly selected minibatch of data to compute the gradient, rather than using the entire dataset.
- It speeds up convergence and helps with large datasets by reducing computation time per update. However, it introduces noise due to random sampling, which doesn't vanish even at minimum

Steps in basic SGD algorithm:

→ Sampling a minibatch from the dataset.

→ Computing the gradient estimate based on this minibatch

$$\text{Gradient estimate} = \hat{g} = \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} L\left(f(x^{(i)}; \theta), y^{(i)}\right)$$

where, $\hat{g}$ → gradient estimate

m → minibatch size

∇_θ → gradient w.r.t parameter θ

L → Loss function

$f(x^{(i)}, \theta)$ → predicted o/p for i/p $x^{(i)}$

$y^{(i)}$ → true label

→ Updating the model parameters by following gradient downhill.

$$\theta \leftarrow \theta - \epsilon \hat{g}$$

ϵ → learning rate

Here, crucial parameter is learning rate ϵ_k which must decrease over time to ensure convergence.

$$\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau \quad \text{where } \alpha = \frac{k}{\tau}$$

$\tau \rightarrow$ iterations

Algorithm: SGD update at training iteration k

Require: Learning rate ϵ_k

Require: Initial parameter θ

while stopping criterion not met do

Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$

Compute gradient estimate:

$$\hat{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}, \theta), y^{(i)})$$

Apply update: $\theta \leftarrow \theta - \epsilon \hat{g}$

end while

* **Momentum:**

- Momentum accelerates SGD by considering past gradients and continuing to move in their direction.
- SGD is popular optimization strategy but sometimes it can be slow hence, SGD with momentum is designed to accelerate learning especially to deal with noisy gradients, small consistent gradients, and high curvature.
- ~~According to Newton laws of motion~~,

- The algorithm accumulates and exponentially decaying average of past gradients using a hyperparameter called decay factor (α) and updates parameters based on both current and past gradients.
- Momentum helps in avoiding oscillations and accelerates convergence.

momentum update rule:

$$v \leftarrow \alpha v - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L(f(x^{(i)}; \theta), y^{(i)}) \right)$$

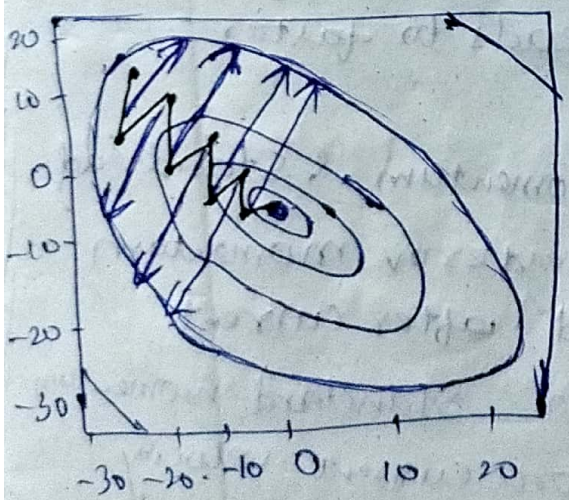
$$\theta \leftarrow \theta + v$$

where $v \rightarrow$ velocity (accumulated gradient)

$\alpha \rightarrow$ decay factor

- we have to decide step size by using terminal velocity

$$\text{Stepsize} = \frac{\epsilon \|g\|}{1-\alpha}$$



Black path → followed by momentum learning Rule

Algorithm: SGD with Momentum:

Require: Learning rate ϵ, momentum parameter α

Require: Initial parameter θ, Initial velocity v.

while stopping criterion not met do

Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots x^{(m)}\}$ with corresponding target $y^{(i)}$.

Compute gradient estimate:

$$g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L\left(f(x^{(i)}; \theta), y^{(i)}\right)$$

Compute velocity update:

$$v \leftarrow \alpha v - \epsilon g$$

Apply update:

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

End while.

---

* Nesterov Momentum:

- Nesterov Momentum is a variant that anticipates the future position of parameters by adding a lookhead step, which leads to faster convergence.
- The diff blw Nesterov Momentum & Standard Momentum is that in Nesterov momentum the gradient is evaluated after current velocity is applied where in standard momentum gradient is evaluated before current velocity applies.

Nesterov momentum update:

$$v \leftarrow \alpha v - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L\left(f((x^{(i)}; \theta) + \alpha v), y^{(i)}\right) \right)$$

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

Scanned with OKEN Scanner

This introduces a lookhead mechanism where the gradient is calculated w.r.t updated weights.

Algorithm: SGD with nesterov momentum

Require: Learning rate ϵ, Momentum parameter α

Require: Initial parameter θ, Initial velocity v

while stopping criterion not met **do**

Sample a minibatch of m examples from the traing set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding target $y^{(i)}$.

Apply interim update: $\tilde{\theta} \leftarrow \theta + \alpha v$

Compute gradient (at interim update):

$$g \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L\left(f(x^{(i)}; \tilde{\theta}), y^{(i)}\right)$$

Compute velocity update:

$$v \leftarrow \alpha v - \epsilon g$$

Apply update:

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

end while

* Parameter Initialization Strategies

Parameter initialization is crucial in training deep learning models, as it significantly impacts convergence, stability and generalization.

- Poor initialization can result in unstable algorithms slow convergence or poor generalization.
- When initializing parameters, especially for hidden units with same activation function, each unit should have different initial parameters to avoid learning identical functions.
- Random Initialization ensures that each unit computes a different function, helping prevent loss of signal in forward propagation and loss of gradient in back propagation.
- The parameters are typically initialized using values drawn from Gaussian or Uniform distribution.

① Weight Initialization:

- weights are usually initialized randomly from Gaussian or uniform distributions.
- Larger intial weights helps to break Symmetry and maintain signal strength during forward and backward propagation.
- Smaller intial weights prevents the exploding gradient problem and avoids activation function Saturation.

Scanned with OKEN Scanner

- One heuristic way to initialize weights is:

$$W = U\left(-\sqrt{\frac{1}{m}}, \sqrt{\frac{1}{m}}\right)$$

where $m \rightarrow$ no. of inputs

$U(\cdot) \rightarrow$ Gaussian distribution

- Normalized initialization:

$$W_{ij} = U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

$m \rightarrow$ no. of i/p $U(\cdot) \rightarrow$ Gaussian distribution.

$n \rightarrow$ no. of o/p

② Bias Initialization:

Biases are usually initialized to zero, but in certain cases:

→ For o/p units, biases can be set based on the expected marginal statistics of o/p

→ For ReLU units, setting a bias to a small positive value (eg 0.1) avoids ReLU saturation at initialization.

③ Other parameter initialization:

→ Sparse initialization, where only a few weights per unit are initialized to non-zero values to avoid making individual weights too small in large layers

→ For variance (precision parameters), a good initialization strategy is setting them to 1 or based on marginal variance in training set.

Scanned with OKEN Scanner

* Adaptive learning Rates Algorithms *

- Adaptive learning Rate algorithms are designed to dynamically adjust the learning rate for each parameter during the training of Neural Network.
- This method helps to improve convergence by automatically adjusting how quickly or slowly the model learns based on feedback from the training process.
- The various Adaptive learning algorithms are,
  1. Ada Grad Algorithm
  2. RMS prop "
  3. Adam "

(1) Ada Grad (Adaptive Gradient Algorithm):

The Ada Grad algorithm adjusts the learning rate for each parameter based on the sum of squares of historical gradients. parameters with large gradients see a rapid decrease in their learning rate, while parameters with small gradients see only a slight reduction.

Working:

- Accumulates Squared gradients of parameters over time.
- Learning rate is inversely proportional to the Square root of the sum of these Squared gradients.

Scanned with OKEN Scanner

Algorithm: AdaGrad

Require: Global learning rate ϵ

Require: Initial parameter θ

Require: Small constant δ, perhaps 10^{-7}, for numerical stability

Initialize gradient accumulation variable $r = 0$

While stopping criterion not met do

Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$

Compute gradient:

$$g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$$

Accumulate squared gradient:

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

Compute update:

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

($\odot$: division & square root applied element wise)

Apply update:

$$\theta \leftarrow \theta + \Delta\theta$$

End while

Advantages:

- Performs well for sparse data
- Adapts learning rates based on parameter's gradients, allowing faster learning where necessary

disadv:

- Learning rate may shrink too quickly, leading to slow convergence
- Not well-suited for DL tasks because the accumulation of squared gradients leads to very small learning rates

Scanned with OKEN Scanner

② RMS Prop (Root Mean Square Propagation):

RMSprop modifies AdaGrad by introducing an exponentially decaying average of past squared gradients, preventing learning rate from shrinking too fast. RMS prop is more suited for non-convex optimization, like traing DNN.

we have 2 algorithms
- → Standard RMS prop
- → RMS prop with Nesterov momentum.

Working:

- Maintains an exponentially weighted moving average of squared gradients, discarding older gradient information.
- This allows the learning rates to remain more consistent throughout training.

Algorithm: RMS Prop

Require: Global learning rate ϵ, decay rate ρ

Require: Intial parameter θ

Require: Small constant δ, usually 10^{-6}, used to stabilize division by small number

Initialize accumulation variable $r = 0$

while stopping criterion not met do

Sample a minibatch of m examples from training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.

Compute gradient:

$$g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$$

Accumulate Squared gradient:

$$r \leftarrow \rho r + (1-\rho) g \odot g$$

Scanned with OKEN Scanner

Compute parameter update:

$$\Delta\theta = \frac{-\epsilon}{\sqrt{\delta + r}} \odot g \quad \left(\frac{1}{\sqrt{\delta+r}} \text{ applied elementwise}\right)$$

Apply update:

$$\theta \leftarrow \theta + \Delta\theta$$

end while

Algorithm : RMS prop with Nesterov Momentum

Require: Global learning rate ϵ, decay ~~rate~~ ρ, momentum coefficient α

Require: Initial parameters θ, Initial velocity v

~~Require~~ Initialize accumulation variable $r = 0$

while stopping criterion not met do

Sample a minibatch of m examples from training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.

Compute interim update: $\tilde{\theta} \leftarrow \theta + \alpha v$

Compute gradient: $g \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(x^{(i)}; \tilde{\theta}), y^{(i)})$

Accumulate gradient: $r \leftarrow \rho r + (1-\rho) g \odot g$

Compute velocity update: $v \leftarrow \alpha v - \frac{\epsilon}{\sqrt{r}} \odot g$ $\left(\frac{1}{\sqrt{r}} \text{ applied elementwise}\right)$

Apply update:

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

end while

Advantages

1. Works well for training DNN
2. More robust in non-convex environment compared to ~~Adagrad~~
3. Addresses the diminishing learning rate problem

Disadv

1. Introduces a new hyper parameter (decay rate) that need to be tuned

(3) Adam (Adaptive Momentum estimation)

Adam combines the benefits of both RMSprop and momentum by using exponentially decaying averages of past gradients (first moment) and past squared gradients (second moment). Adam also introduces bias correction to improve the estimates of the moments early in training.

Working:

- Computes the exponentially weighted moving average of both the gradients and the squared gradients.
- Applies bias correction to both estimates to account for their initialization at zero.
- Updates the parameters based on both momentum & adaptive learning rates.

Algorithm: Adam

Require: step size (ϵ) (suggested default: 0.001)

Require: Exponential decay rates for moment estimates ρ_1 & ρ_2 in $[0, 1)$ (defaults: 0.9 and 0.999)

Require: Small constant δ used for numerical stabilization (default: 10^{-8})

Require: Initial parameters θ

Initialize 1st and 2nd moment variables $s = 0$, $r = 0$

Initialize time step $t = 0$

while stopping criterion not met do

Sample a minibatch of m examples from the training set $\{x^{(1)}, \dots x^{(m)}\}$ with corresponding targets $y^{(i)}$

Compute gradient:

$$g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L\left(f(x^{(i)}; \theta), y^{(i)}\right)$$

~~$t \leftarrow t+1$~~

Update biased first moment estimate:

$$s \leftarrow \rho_1 s + (1-\rho_1) g$$

Update biased second moment estimate:

$$r \leftarrow \rho_2 r + (1-\rho_2) g \odot g$$

Correct bias in first moment:

$$\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$$

Correct bias in second moment:

$$\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$$

Compute update: $\Delta\theta = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}} + \delta}$ (element wise applied)

Apply update:

$$\theta \leftarrow \theta + \Delta\theta$$

end while

Adv

- Most commonly used optimizer in DL
- Very efficient at handling noisy & sparse gradients
- Bias correction improves performance during initial stages of training

disAdv

- May lead to overfitting
- Sometimes require rate tuning

* Approximate Second-order Methods *

We have three optimization algorithms for deep network training that leverage second-order information those are:

- Newton's Method
- Conjugate Gradient.
- BFGS (Broyden-Fletcher-Goldfarb-Shanno).

Consider the objective function:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} L(f(x^{(i)}; \theta), y^{(i)})$$

① Newton's Method:

Newton's method uses a second-order Taylor Series approximation of objective function $J(\theta)$ to iteratively update model parameters. The update rule for the parameters is:

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

where

$H \rightarrow$ HessianMatrix (Second derivatives w.r.t θ)

$\nabla_\theta J(\theta_0) \rightarrow$ gradient of objective function at θ_0

Newton's method converges quickly for quadratic functions but it is computationally expensive due to Hessian inversion.

Algorithm: Newton's method

Require: Initial parameters θ

Require: Traing set of m examples

while stopping criterion not met do

Compute gradient:

$$g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$$

Scanned with OKEN Scanner

compute Hessian:

$$H \leftarrow \frac{1}{m} \nabla_\theta^2 \sum_i L\left(f(x^{(i)};\theta), y^{(i)}\right)$$

compute Hessian inverse: H^{-1}

compute update: $\Delta\theta = -H^{-1}g$

Apply update:

$$\theta \leftarrow \theta + \Delta\theta$$

endwhile

② Conjugate Gradient:

Conjugate gradients is a method to efficiently avoid the calculation of the inverse Hessian by iteratively descending conjugate direction.

It iterates over search direction that are conjugate to previous directions, ensuring the progress made in earlier iterations is not undone.

Updation rule:

$$d_t = \nabla_\theta J(\theta) + \beta_t d_{t-1}$$

where, $\beta_t \rightarrow$ coefficient that determines how much of previous direction d_{t-1} should be incorporated into the current search direction.

two common formulas for β_t are:

(a) Fletcher-Reeves: $\beta_t = \dfrac{\nabla_\theta J(\theta_t)^T \nabla_\theta J(\theta_t)}{\nabla_\theta J(\theta_{t-1})^T \nabla_\theta J(\theta_{t-1})}$

(b) Polak-Ribiere: $\beta_t = \dfrac{(\nabla_\theta J(\theta_t) - \nabla_\theta (J(\theta_{t-1}))^T \nabla_\theta J(\theta_t)}{\nabla_\theta J(\theta_{t-1})^T \nabla_\theta J(\theta_{t-1})}$

Conjugate gradient works well in quadratic settings, but for non-quadratic function we use Non-linear Conjugate Gradients.

Scanned with OKEN Scanner

3) BFGS:

BFGS (Broyden - Fletcher - Goldfarb - Shanno) is a quasi-Newton method that approximates the inverse Hessian Matrix without calculating it directly.

- BFGS iteratively refines an approximation to H^{-1}, denoted as M_t, using low-rank updates.

The update rule for the parameters is:

$$\theta_{t+1} = \theta_t + \epsilon^* p_t$$

where $p_t = M_t g_t$ is search direction

ϵ^* = step size determined by linesearch.

BFGS brings benefits of second-order methods without computational overhead of explicitly computing Hessian.

The L-BFGS (Limited memory version of BFGS) is often used for large scale problems as it requires significantly less memory by storing only a few vectors that approximate the inverse Hessian.

Scanned with OKEN Scanner

Convolutional Networks:

def: Convolutional networks also called as Convolutional Neural Network (CNN) introduced by LeCun in 1989, are specialized neural network for processing grid-like data (eg: images and time-series)

here, In CNN we use a mathematical operation called Convolution instead of general matrix multiplication in atleast one layer.

CNN has tremendous application in image processing.

* Convolution Operation *

- **Convolution:** Convolution is a mathematical operation that takes two functions and combines them to produce a third function (smoothed function)

This operation is commonly used in DL in CNN to extract features from input data like images

The General formula of **Continous Convolution** is given by

$$S(t) = \int_{-\infty}^{\infty} x(a)\, w(t-a)\, da$$

$x(t) \rightarrow$ i/p function such as time-series or image data

$w(t) \rightarrow$ Kernel (filter), used to modify the input by focusing on specific features like edges or patterns

$S(t) \rightarrow$ o/p function, reffered as feature map (or) activation map

The General formula for discrete convolution is

$$S(t) = \sum_{a=-\infty}^{\infty} x(a)\, w(t-a)$$

Since most data in NN is digital, convolution is often implemented in a discrete form.

The most general way to show convolution as:

$$S(t) = x(t) * w(t)$$

$$S(t) = (x * w)(t)$$

- **Multidimensional Convolution:**

In image processing, data is often represented as 2D grid (pixel arrays) so, we extend convolution to 2D.

For 2D Image I and 2D Kernel K, the discrete convolution is

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)\, K(i-m, j-n)$$

where

- $I(m,n)$ is value of i/p image at coordinates (m,n)
- $K(i-m, j-n)$ is Kernel

- Convolution is commutative in nature that is, the order of functions can be swapped

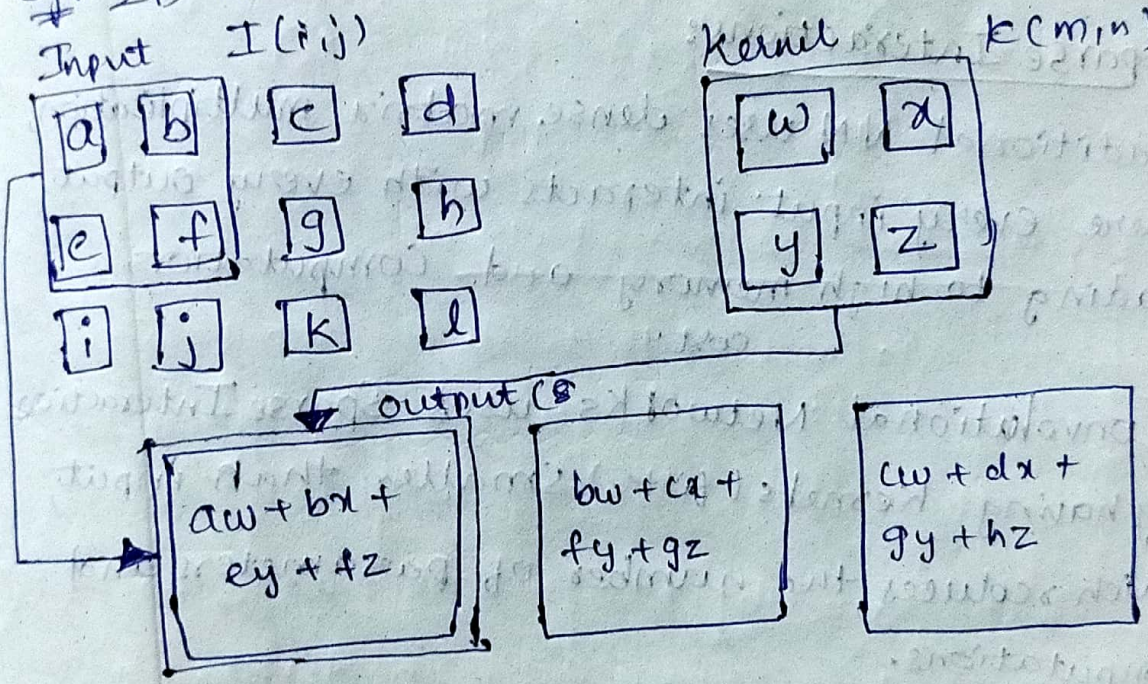
Scanned with OKEN Scanner

Cross-correlation is same as convolution but in cross-correlation is differ from convolution by not flipping the Kernel.

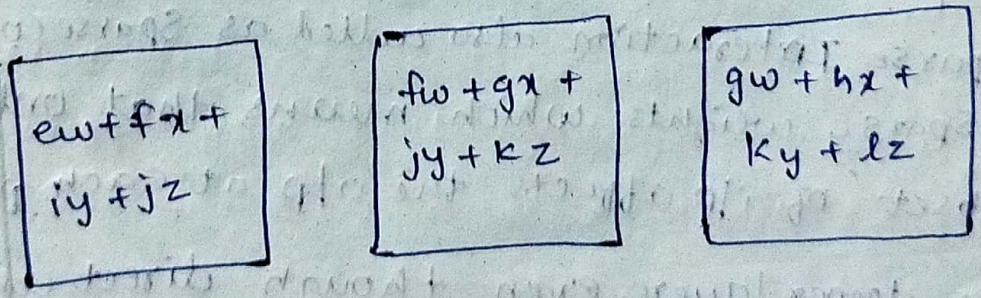
The formula for cross-correlation is:

$$S(i,j) = \sum_m \sum_n I(i+m, j+n)\, K(m,n)$$

eg: 2D convolution without Kernel-flipping



S(i,j)



Scanned with OKEN Scanner

Motivation

The Three important ideas behind convolution that improve ML systems are:

- Sparse interaction
- parameter sharing
- equivariant Representation

These concepts help CNN's process inputs more efficiently and effectively.

① Sparse Interactions:

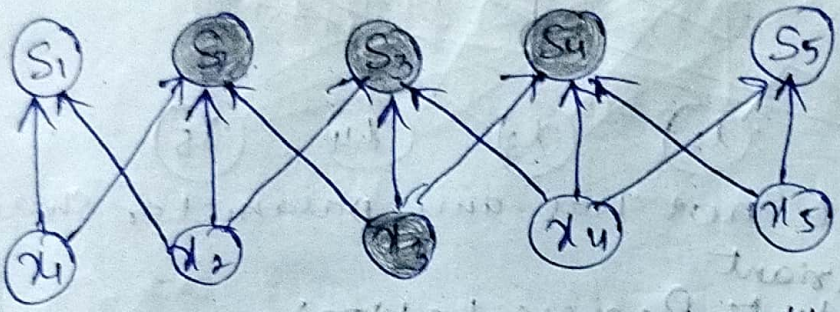
- Traditional NN uses dense matrix multiplication, where every input interacts with every output, leading to high memory and computation cost.
- Convolutional Networks uses Sparse Interaction by having Kernels (filter) smaller than input, which reduces the number of parameters and computations.
- Sparse Interaction also called as Sparse connectivity or sparse weights which means that only a subset of i/p affects the o/p at each position
- In deeper layers, even though direct connections remain sparse, units may indirectly interact with a larger portion of input, allowing network to capture complex relationships.

Eg: detecting edges in images can be done with small kernels, even though images may have millions of pixels.

Scanned with OKEN Scanner

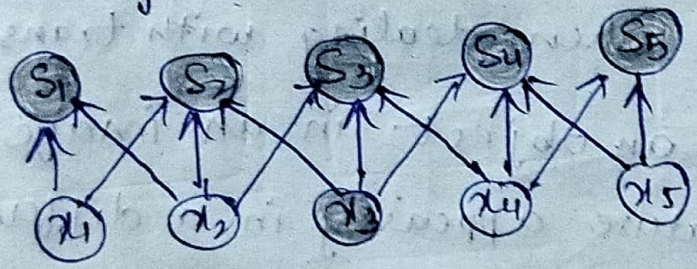
eg: Sparse connectivity, viewed from below

One input unit x_3 and highlighted o/p units S affected by x_3



when S is formed by convolution with a kernel of width 3, only three o/p s are affected by x.

when S is formed by Matrix Multiplication, connectivity is no longer sparse, all of o/ps are affected by x_3

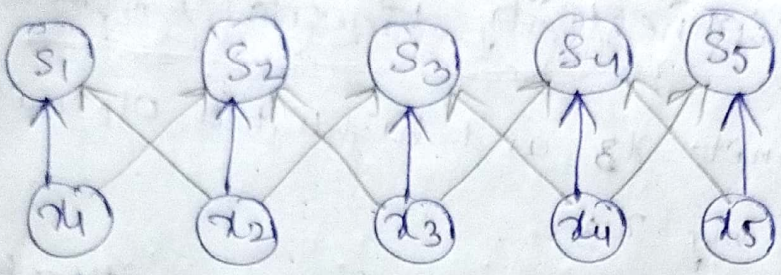


② Parameter Sharing:

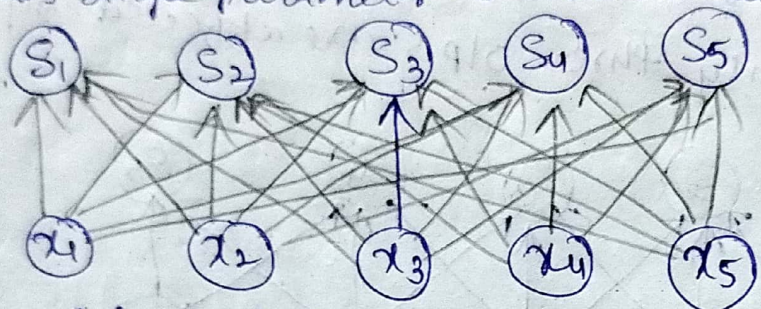
- Parameter Sharing means the same kernel parameter are used across multiple location in the input. this reduces the no. of parameters needed.
- This makes convolutional networks more memory-efficient than dense networks, reducing the number of parameters to k, where k is kernel size.

Scanned with OKEN Scanner

eg:



only central element of 3-element kernel in convolutional model, this single parameter is used at all i/p locations



This model not has any parameter sharing.

(3) Equi~~extent~~ variant Representation:

- Equivariant means that if the input shifts, the output shifts in same way. Convolution has this property when dealing with translation.

eg: Shifting an object in an image will result in same feature appearing in a different position in the output. This makes convolution useful for detecting features that may appear in different locations, such as edges in images.

* Optimization Strategies and Meta Algorithms

Optimization strategies refers to techniques and methods used to find the best set of parameters (weight & bias) in ML, such that the model's performance is maximized (error minimized)

The goal is to minimize a lossfunction by adjusting the parameters during traing

Meta Algorithms are higher-level strategies designed to make the learning process itself more efficient.

(1) Gradient descent:

the first order optimization that adjusts model's parameters in direction of the negative gradient of the lossfunction to minimize error.

→ Stochastic GD - updates parameters using a random single data pt

→ Mini batch GD - updates parameters using small subset of data pts

→ Batch GD - update, parameters using entire data pts.

(2) Batch Normalization:

Batch normalization stabilizes and speeds up training of DNN by normalizing the input to each layer so that it has zero mean and unit variance.

Scanned with OKEN Scanner

Normalization: $H^1 = \frac{H-\mu}{\sigma}$

where

$H \rightarrow$ mini batch Activation

$\mu \rightarrow$ mean of activation for each unit

$$\mu = \frac{1}{m}\sum_i H_i$$

$\sigma \rightarrow$ SD of activation for each unit

$$\sigma^2 = \frac{1}{m}\sum_i (H_i - \mu)^2 + \delta$$

$m \rightarrow$ no. of training eg

$\delta \rightarrow$ constant to prevent division by zero

(3) Polyak Averaging: This method involves averaging multiple points ~~visited during~~ in trajectory optimization to Smooth out fluctuation and improve convergence

Averaging of trajectory

$$\hat{\theta}(t) = \frac{1}{t}\sum_i \theta(i)$$

exponentially decaying Running Avg:

$$\hat{\theta}(t) = \alpha\,\hat{\theta}(t-1) + (1-\alpha)\,\theta(t)$$

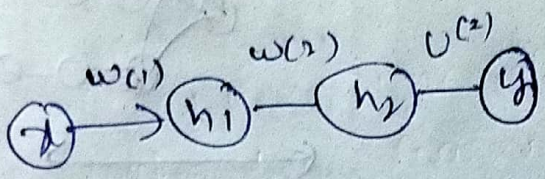
$\alpha \rightarrow$ decay factor.

Scanned with OKEN Scanner

) Supervised pretraining: It involves training simpler models or tasks before training the complex models"

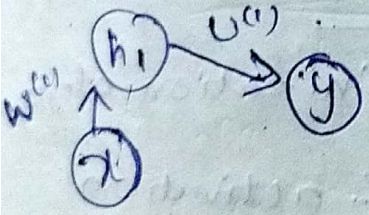
· Greedy supervised pretraining: It firstly breaks a given supervised learning problem into simpler supervised learning and solves then to get optimal version of result

eg:

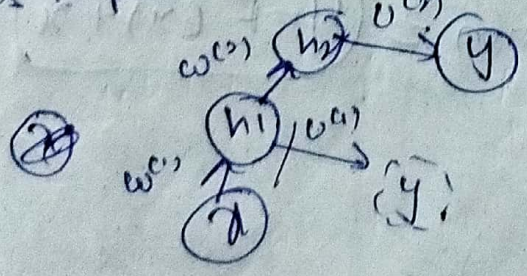


To train this using greedy

1) first we train 1st hidden layer using x and to get y



2) To train second hidden layer we remove o/p layer from previous network & give it as input to next layer



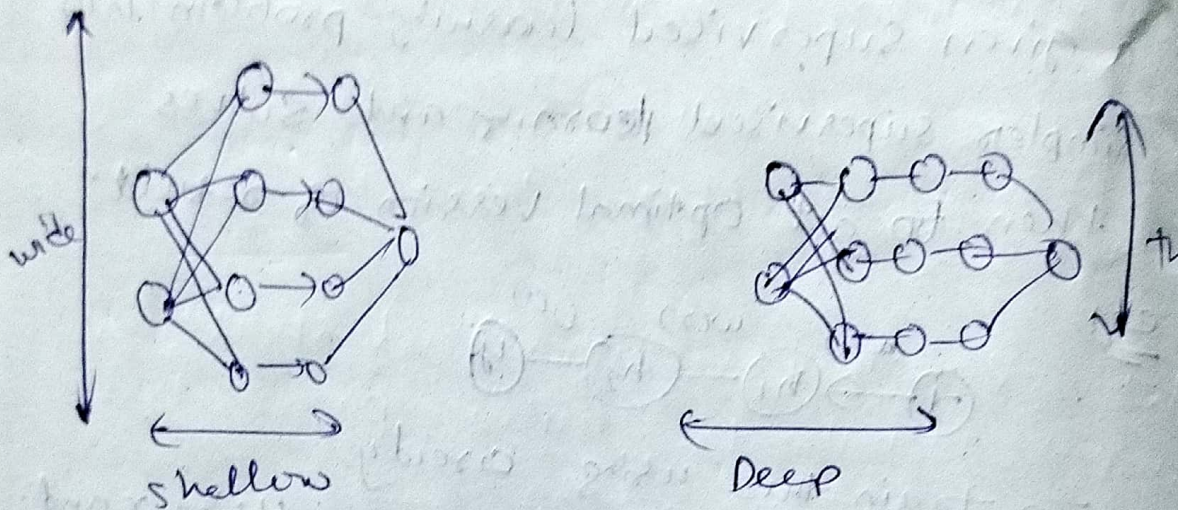
· Fit Nets:

here, the shallow and wider teacher Nw is trained first then second network which is thinner & deeper called student nw is trained.

And use the student n/w to predict the intermediate & final layers of teacher n/w

Teacher Nlw

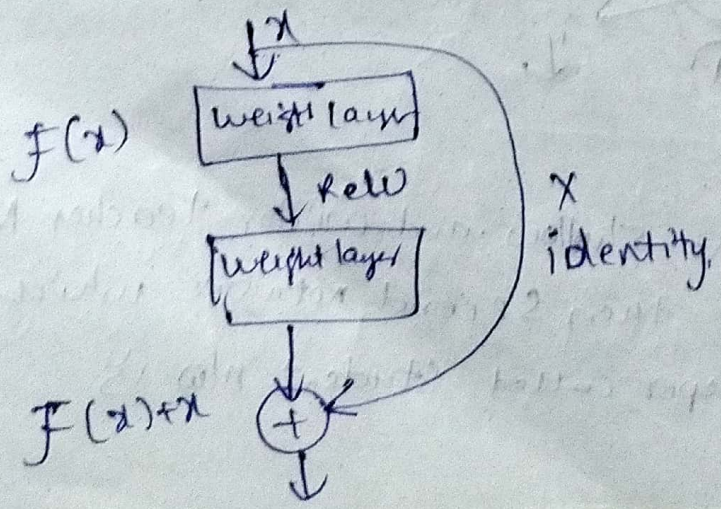
Student Nlw



(8) (2) Designing model to Aid optimization

Instead of developing new optimization algo, we use linear Transformation & activation functions like ReLU to reduce the complexity of optimization.

Residual connection = $y = F(x) + x$



Pooling

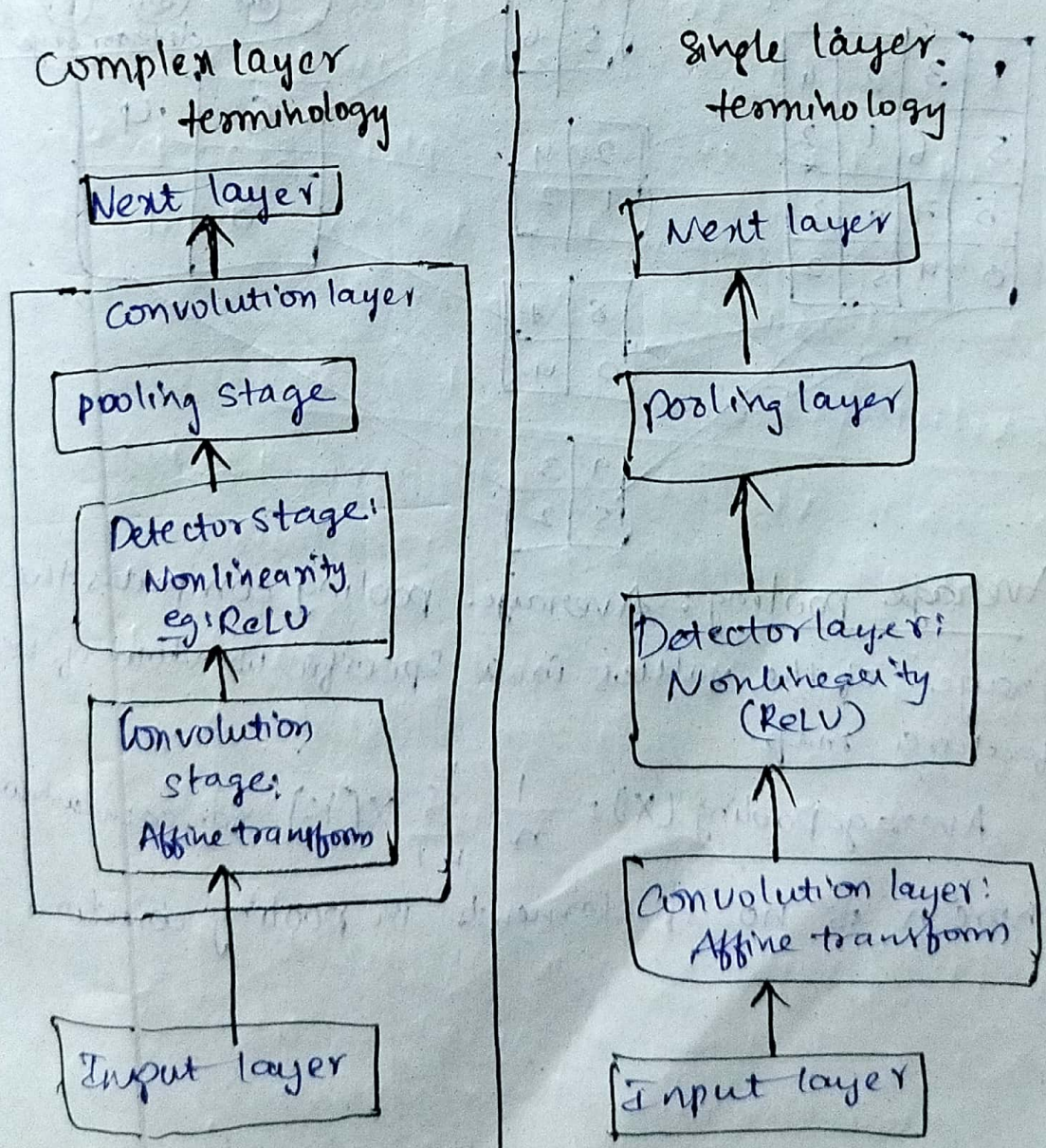
⇒ Stages in CNN:

A typical CNN layer involves three stages.

① convolution stage: produces linear activations by applying filters to the input.

② Detector stage (Nonlinear Activation): Applies an activation function (ReLU) to each activation.

③ Pooling Stage: Modifies the output by summarizing nearby activations through pooling functions.



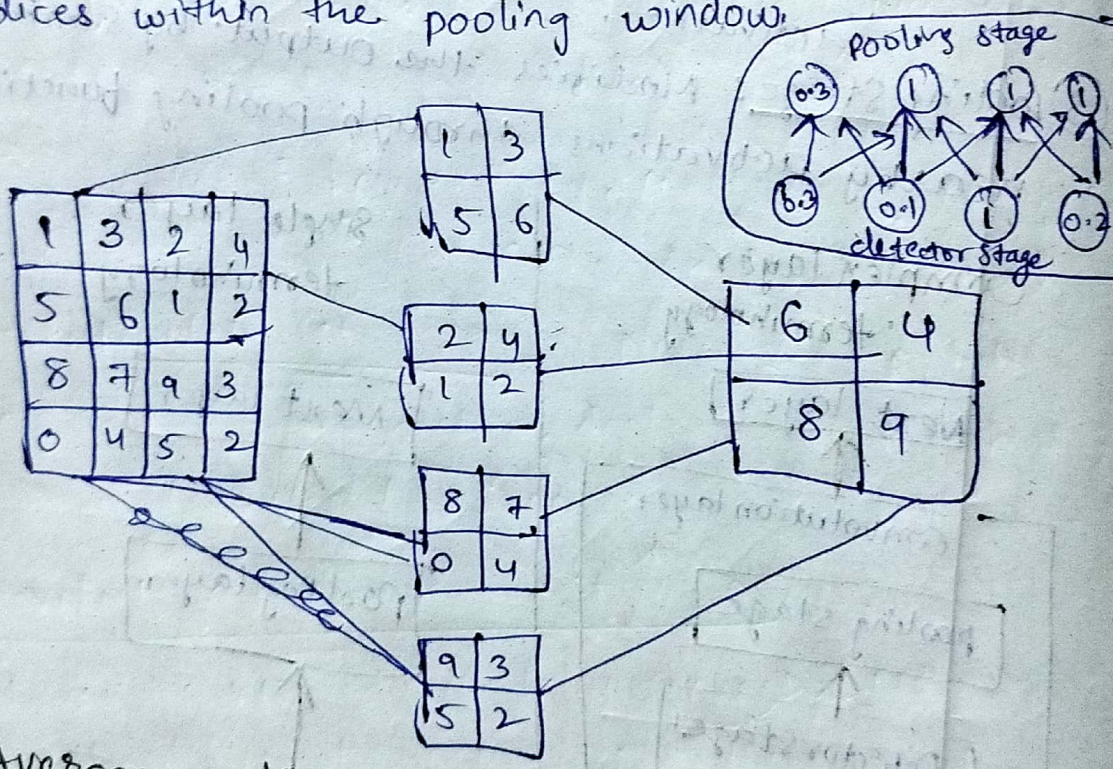
Scanned with OKEN Scanner

⇒ Pooling Functions: These functions replace outpu at specific locations with summary statistics

① Max pooling: Max pooling selects the maximum value from a specific window (or region) of the feature map.

MaxPooling(X) = max(X[i,j]) for each window where X is the input feature map and i,j are indices within the pooling window.

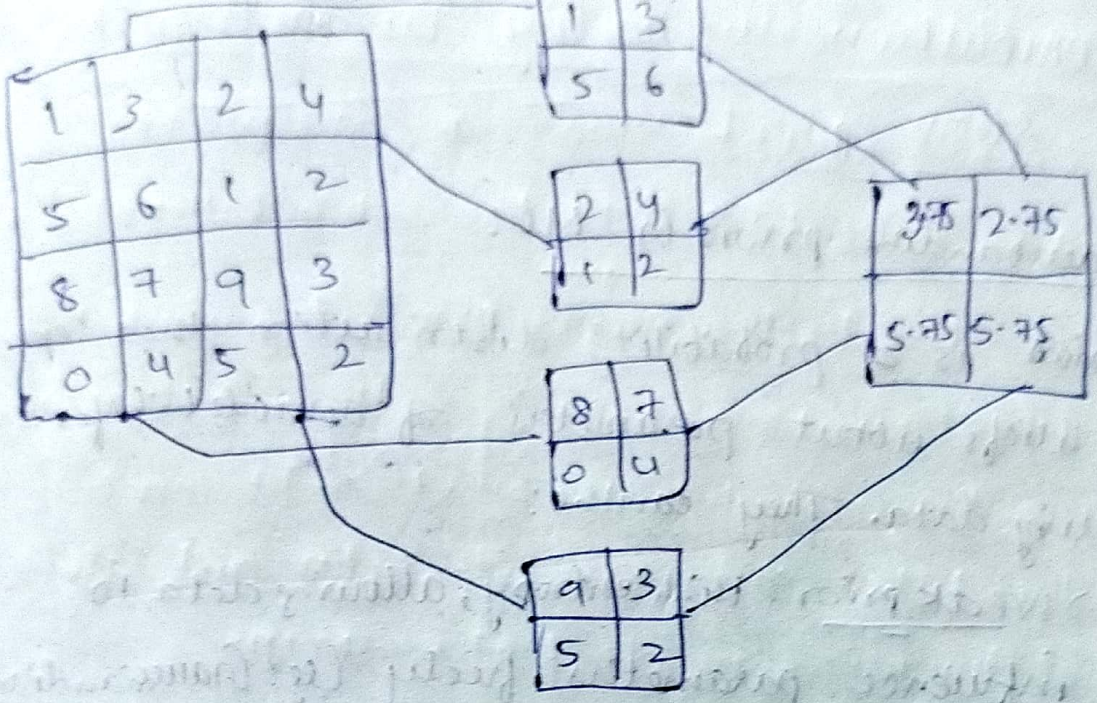
eg:



② Average pooling: Average pooling computes the average of the values in a specific window of the feature map.

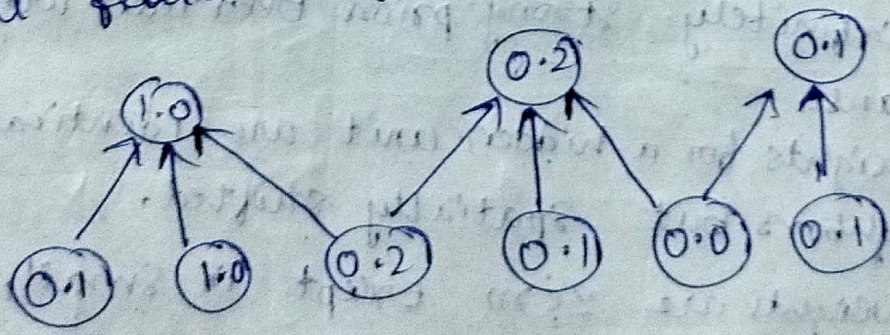
$$\text{AveragePooling}(X) = \frac{1}{n} \sum_{k=1}^{n} X[k]$$ for each window

where n is no. of elements in pooling window



⇒ Invariance: pooling enhances translation invariance meaning the pooled o/p remains stable even if the inputs shifts slightly. this is particularly used in tasks like object detection.

⇒ pooling with downsampling: This shows how reducing the representation size helps reduce computational requirements while still capturing the essential features of the input.



* Convolution and pooling as an Infinitely strong prior:

⇒ Understanding priors in ML:

A prior is a probability distribution that represents our beliefs about parameters of a model before observing data. They can be:

→ Weak prior: High entropy, allowing data to influence parameters freely (eg: Gaussian distribution with high variance)

→ Strong prior: Low entropy, constraining parameters significantly (eg: GD with low variance)

→ Infinitely strong prior: places zero probability on certain parameters, declaring them forbidden regardless of data support.

⇒ Convolution Networks as strong priors:

- CNN can be viewed as fully connected networks with infinitely strong priors over their weights

this means:

→ weights for a hidden unit are identical to its neighbors, but spatially shifted.

→ weights are zero except in small, spatially contiguous receptive fields.

- Infinitely strong prior means that convolutional layers impose strong assumptions about the structure of data emphasizing local interactions and translations invariance.

Scanned with OKEN Scanner

Similarly, pooling layers are seen as imposing strong priors that encourage invariance to small translations in the input data.

* Variants of Basic convolution function:

The various variants of convolution function are:

→ parallel Application of convolution / convolution without out flipping

- In NN, convolution is typically applied in parallel with multiple kernels to extract various features simultaneously from i/p data, rather than a single kernel that captures one type of feature.

- The input to convolutional layers is often a grid of vector-valued observations (images) therefore operations are conducted on 3-D tensors (channels, rows & columns). and often organised as 4D-tensors for batch processing.

- ~~The~~ let 3-D kernel tensor as Z and 4-D tensor K with elements $K_{i,j,k,l}$ where i,j are input and output and k,l are rows & columns then convolution operation is given by:

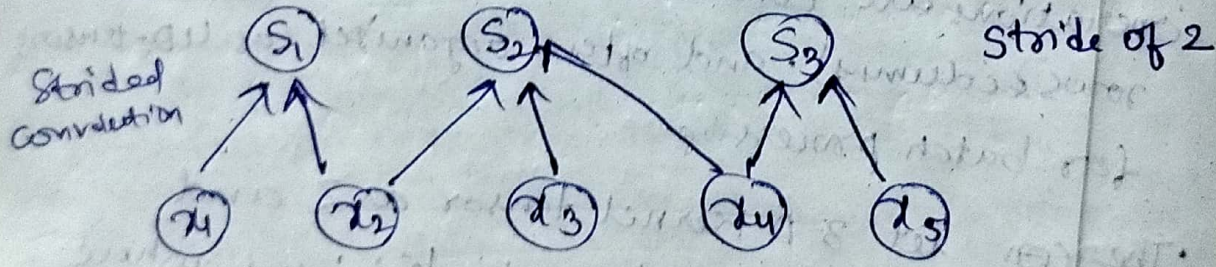
$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m,k+n} K_{i,l,m,n}$$

This emphasizes that each element of o/p is a weighted sum of i/p element adjusted by kernel K

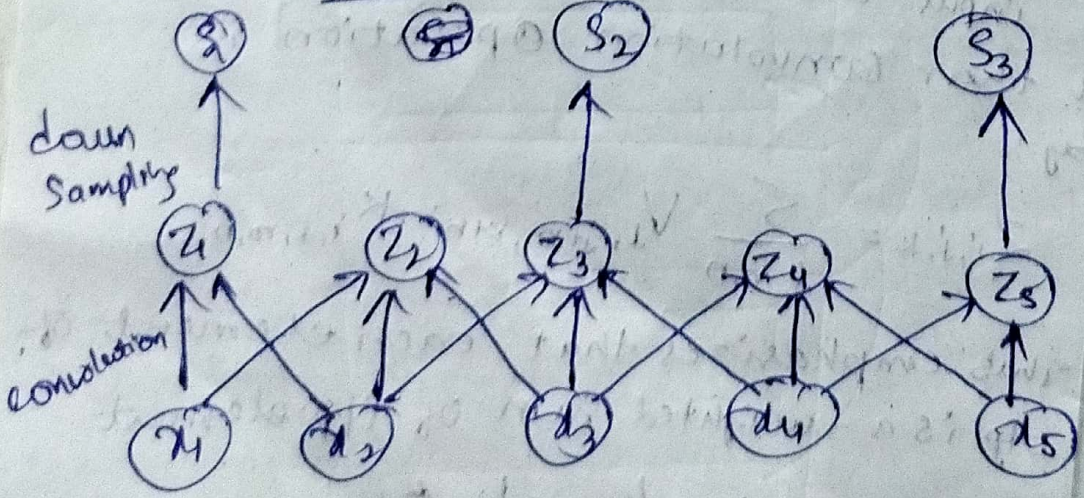
⇒ Convolution with downsampling and strides:

- downsampling refers to skip over some positions of kernel in order to reduce the computational cost.
- Stride refers to the no. of pixels by which the kernel / filter is moved accross the input image during the convolution operation. It controls how the filter slides over i/p data and affects spatial dimensions of o/p feature map.
  - stride of 1 means filter moves 1 pixel at a time, both horizontally & vertically.
  - A stride of 2 means filter moves 2 pixel at a time reducing o/p size.

Stride convolution:



Downsampling convolution:



Scanned with OKEN Scanner

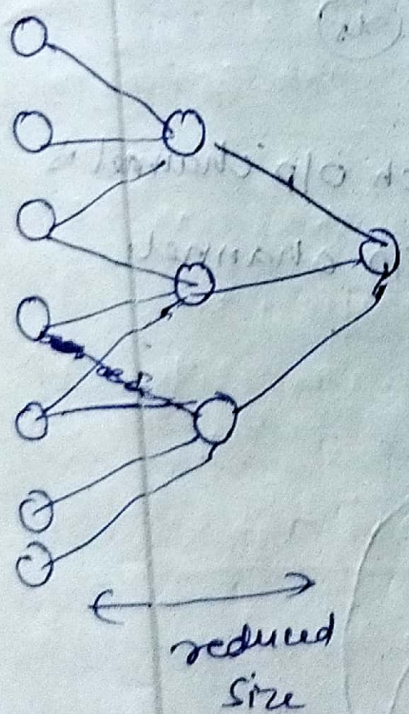
⇒ Zero padding: Zero padding is a critical feature that allows the input to be wider, controlling output size independently from kernel width. Different zero padding strategies lead to different convolution types:

→ Valid convolution: No padding, leading to output size shrinkage.

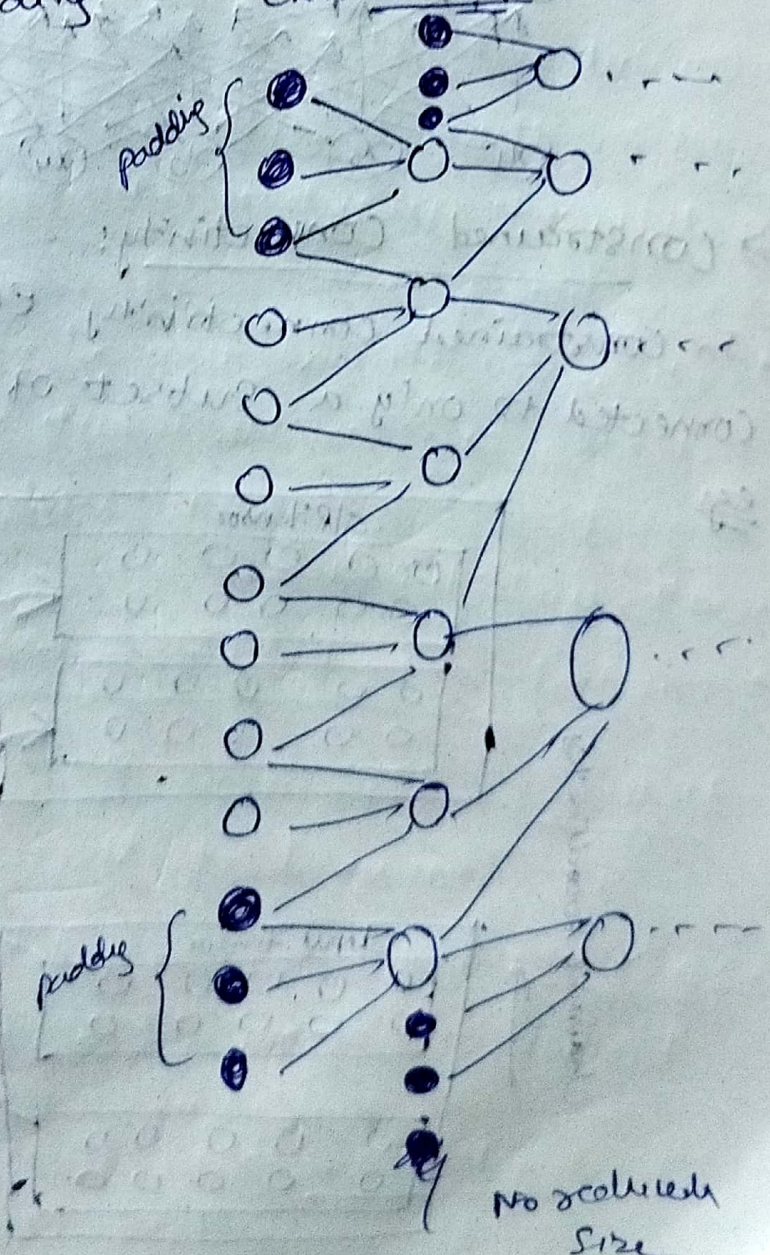
→ Same Convolution: padding added to maintain i/p & o/p sizes

→ full convolution: Maximum padding, which increases o/p size.

CNN without padding

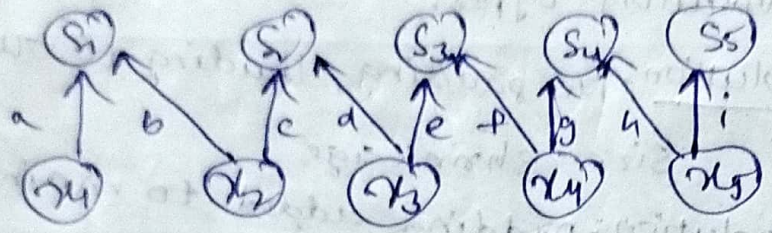


CNN with padding

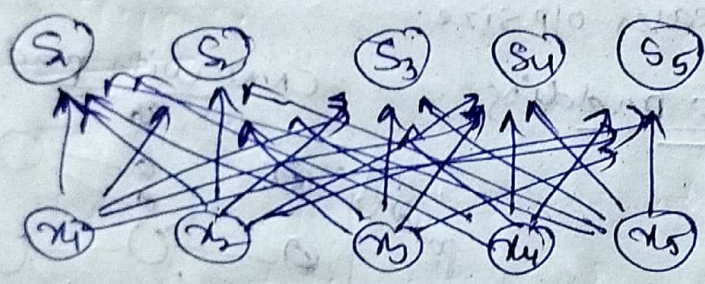


⇒ Local connections, convolution & full connections

→ Local connections use distinct weights for each spatial locations as opposed to sharing weights in standard convolutions



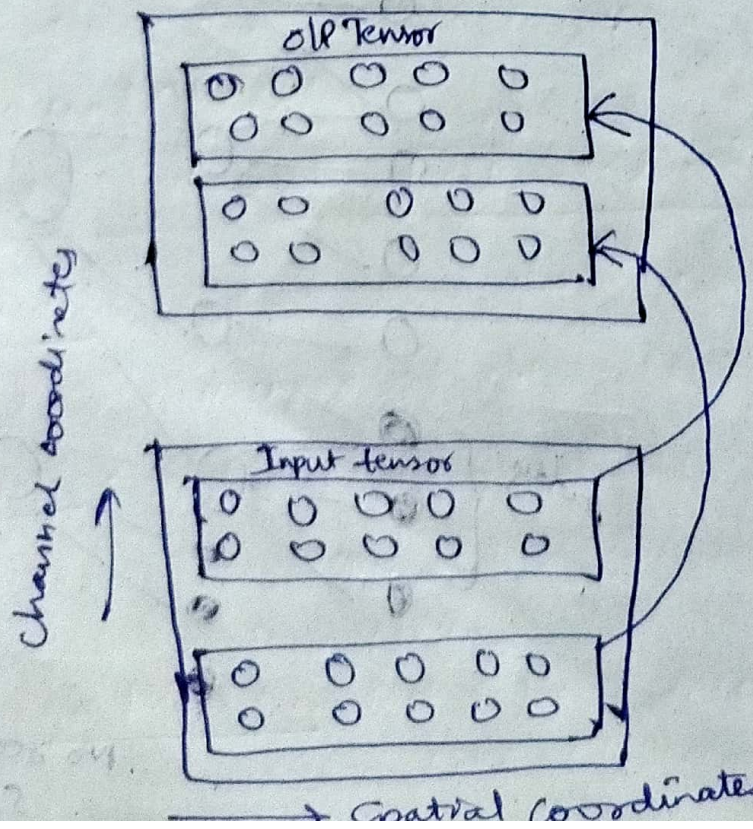
→ Full connections where ~~each~~ every node is connected with each others



⇒ Constrained Connectivity:

In constrained connectivity each o/p channel is connected to only a subset of i/p channels.

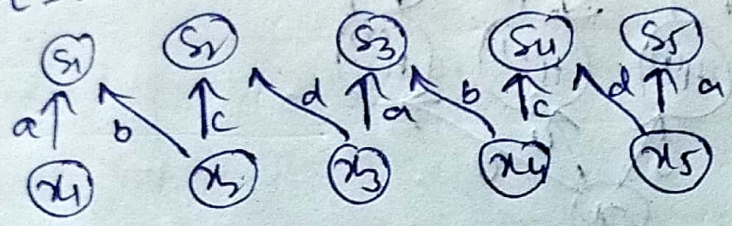
eg:



⇒ Tiled convolution

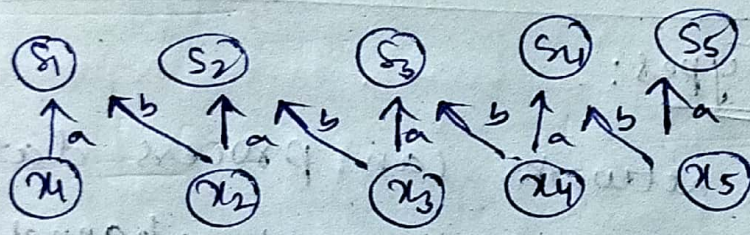
→ This approach strikes a balance blw convolutional and locally connected layers by rotating through a set of kernels rather that using distinct weights at every positions as in local connectivity

Eg: $t = 2$



ab } two sets
cd } of wts

$t = 1$



a b } one set

* Structured outputs:

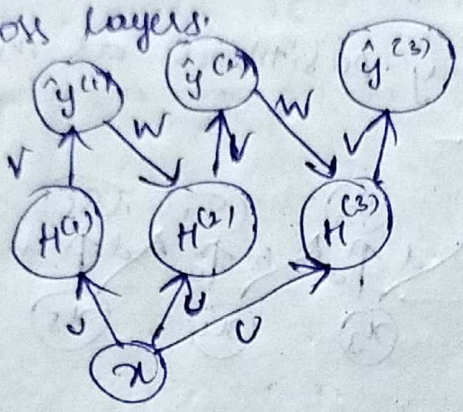
- Convolutional networks can produce structured outputs such as tensor that represents probability belonging to different class, instead of just class labels or real values
- The o/p tensor S indicates the probability $S_{i,j,k}$ that pixel (j, k) belongs to class i, allowing for precise segmentation of images and drawing masks around individual objects.

• Strategies to Maintain o/p size:

→ Avoid pooling: Not using pooling layers can help maintain the spatial dimensions similar to i/p.

→ Unit stride pooling: Using pooling operator with a stride of 1 can help to preserve spatial dimensions

→ Recurrent Convolutional Networks: An initial estimate of pixel labels can be refined iteratively using neighboring pixels leading to RNN that shares weights across layers.



Data Types:

- Convolutional networks can process data with multiple channels, where each channel represents a different quantity observed over space or time.
- Unlike traditional neural networks, convolutional networks can handle input varying spatial dimensions, such as images with different widths and heights.
- Convolution with Different Data Types:

- 1D Audio: Time series data where amplitude is measured at discrete time steps.
- 2D Audio: Audio transformed into 2D tensor via Fourier Transform, with frequency rows & time cloums
- 3D-volumetric data: Data from medical imaging Technologies like CT scans

Scanned with OKEN Scanner

Data Formats for CNN

Datatype	Description
Single channel	1-D Audio: Amplitude measurements over time
	2D-Audio: Fourior Transformed audio with Frequency & time dimensions
	3D-volumetric Data: Data from medical imaging, such as CT scans.
Multi-channel	Skeleton Animation: Character pose data with joint angles over time
	Color Images: RGB pixel channels in 2D imgs, enabling translational invariance
	Color video: Dimensions include time, height & width of video frames.

* Efficient Convolution Algorithms:

In modern convolutional networks, contains over a million units and require powerfull implementations with parallel computing.
The various techniques to accelerate convolution are:

Fourier Transform convolution:

convolution can be ~~speed up long~~ accelerated by transforming inputs and kernels to the fequency domain, Performing pointwise multiplication and converting back to time domain.

Scanned with OKEN Scanner

Using Fourier transforms approach is faster than direct convolution in certain cases.

Separable Kernels:

- A kernel is separable if it can be expressed as the outer product of 1D vectors, one for each dimension.
- Instead of performing a single d-dimensional convolution, one can achieve significant speedup by executing multiple 1-D convolutions
- Separable kernels enable faster computation by reducing d-dimensional convolutions to multiple 1-D convolutions, decreasing both runtime and storage requirement

→ Naive Multidimensional Convolution requires ~~O(w^d)~~ $O(w^d)$ time & space, where w is kernel size in each dimension (d)

→ Seperable convolution: Requires $O(w \times d)$ time & space, leading to substantial saving in both computation time & parameter storage.

Random or Unsupervised Features

Training convolutional networks is costly due to the computational demands of feature learning. This expense primarily comes from repeated fwd & bwd propagation steps involved in supervised gradient-based learning.

Use of Random filters: The networks with randomly assigned weight can effectively mimic selective filters after pooling, providing low-cost way to evaluate network architectures.

Greedy Layer-wise Pretraining: Another approach involves greedy layer-wise pretraining, which trains each layers independently and sequentially. This method extended from multilayer perceptrons, minimizes the need for repeated Backpropagation at each step.

Unsupervised Feature Learning: To reduce cost, feature learning in convolutional networks can be unsupervised. There are 3 methods for this those are:

① **Random Initialization:** Kernels can be randomly initialized, often achieving effectiveness in extracting frequency-selective and translation-invariant features.

② **Hand-Designed Kernels:** Predefined Kernels, such as edge detectors, can be manually specified.

③ **Unsupervised Learning with K-means:** Techniques like K-means clustering can create convolution kernels by grouping similar image patches into

Centroids used as Kernels. This method Separates feature learning from classifier training allowing feature extraction in a single step for entire dataset

- Unsupervised pretraining was common strategy to manage computational limitations and small datasets. This may add regularization benefits or support larger architectures due to reduced training cost.

The Neuroscientific Basics of Convolution Networks

- Convolutional Networks are inspired by neuroscientific principles
- The human primary visual cortex (V1) processes visual information by spatial mapping and feature detection. CNN adopts similar mechanisms, such as feature maps, localized receptive fields etc.

eg: Receptive field for a neuron can be mathematically represented as:

$$R(x,y) = \sum_i \sum_j I(x+i, y+j) \cdot K(i,j)$$

$I(x,y) \rightarrow$ i/p image $K(i,j) \rightarrow$ kernel filter

- In V1, simple cells detect localized features, while complex cells achieve position invariance inspiring CNN pooling layers. This can be implemented as:

$$P(x,y) = \max_{i,j} (R(x+i, y+j))$$

Scanned with OKEN Scanner

• Grand

↑ High visual pathways, like inferotemporal cortex (IT) contain "grandmother cells" that activate for specific objects or concepts. In CNNs, later layers similarly capture complex patterns across transformations.

⇒ Biological vs. convolutional Networks:

The several distinctions blw CNN & human visual system are:

→ Human vision relies on high-resolution Foveal focus, whereas CNN's process entire image at once.

→ Human vision is multi-sensory and context-aware, while CNNs are limited to visual data.

→ Feedback in human vision contrasts with CNN's largely feedforward structure.

Chapter -3

Sequence Modeling: Recurrent and Recursive Nets:

Recurrent Neural Networks specializes in processing sequential data by handling variable-length sequences and maintaining dependencies across time steps.

- Parameter sharing across time steps enables RNNs to generalize across sequence lengths and capture patterns that may appear in any position.
- Temporal convolution and time delay networks provide parameter sharing over time, but RNN's uses a recurrent structure to capture sequential dependencies.
- Cycles in computational graph allows RNNs to retain information from past time steps, supporting long-term dependencies.
- Mini-batch processing with flexible sequence length allows RNNs to handle variable length data in practical applications.

---

* Unfolding Computational Graphs *

Unfolding a computational Graphs is a process that transforms a recursive or recurrent computation into a graph with a repetitive structure, commonly seen in sequence-based models. This unfolding includes parameter sharing, leading to a deeper network structure that can model sequences of events.

Scanned with OKEN Scanner

• A recurrent Dynamic System describes how a system's state evolves over time. Typically, the state at time t (denoted as $s(t)$) depends on the previous state $s(t-1)$. The following is a recurrence Relation:

$$s(t) = f(s(t-1); \theta) \quad \text{---} (1)$$

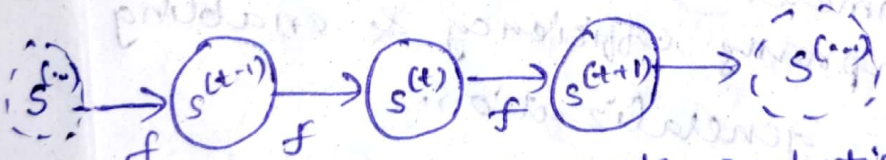
This eqn shows a classical dynamical system, where the function f updates the system's state using parameters θ that remain the same at each time step.

unfolding eqn (1)

If we apply this relation over a finite number of time steps T, we get an expanded form that no longer involves recurrence eg: $T = 3$

$$s(3) = f(f(s(1); \theta); \theta)$$

The result is directed acyclic computational graph, where each state depends directly on previous state



This graph illustrate evolution of states $s(t-1)$, $s(t)$, $s(t+1)$ over time.

In more complex systems, an external input $x(t)$ influences each state:

$$s(t) = f(s(t-1), x(t); \theta)$$

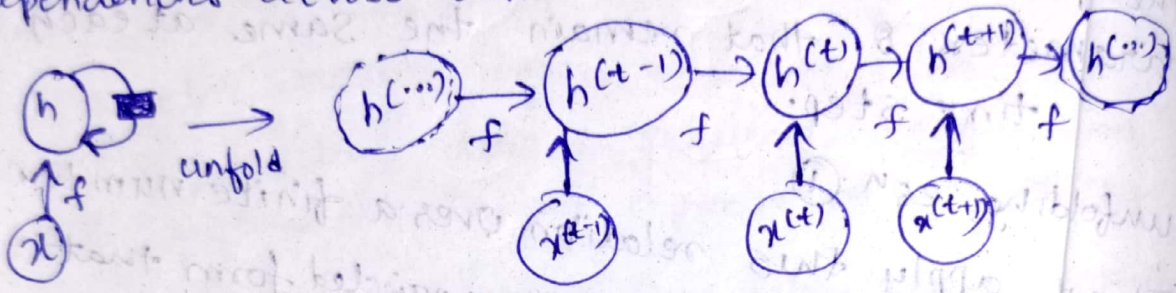
Here state incorporates information from both previous state & external input, for more complex representations in Sequence modeling

Recurrent NN:

RNN process sequences by maintaining a "hidden state" $h(t)$, which serves as a summary of the information received so far:

$$h(t) = f(h(t-1), x(t); \theta)$$

Here, hidden state $h(t)$ allows network to learn dependencies across time steps.



Advantages of Unfolding:

① **Fixed model size:** Regardless of sequence length, the model's input size remains the same.

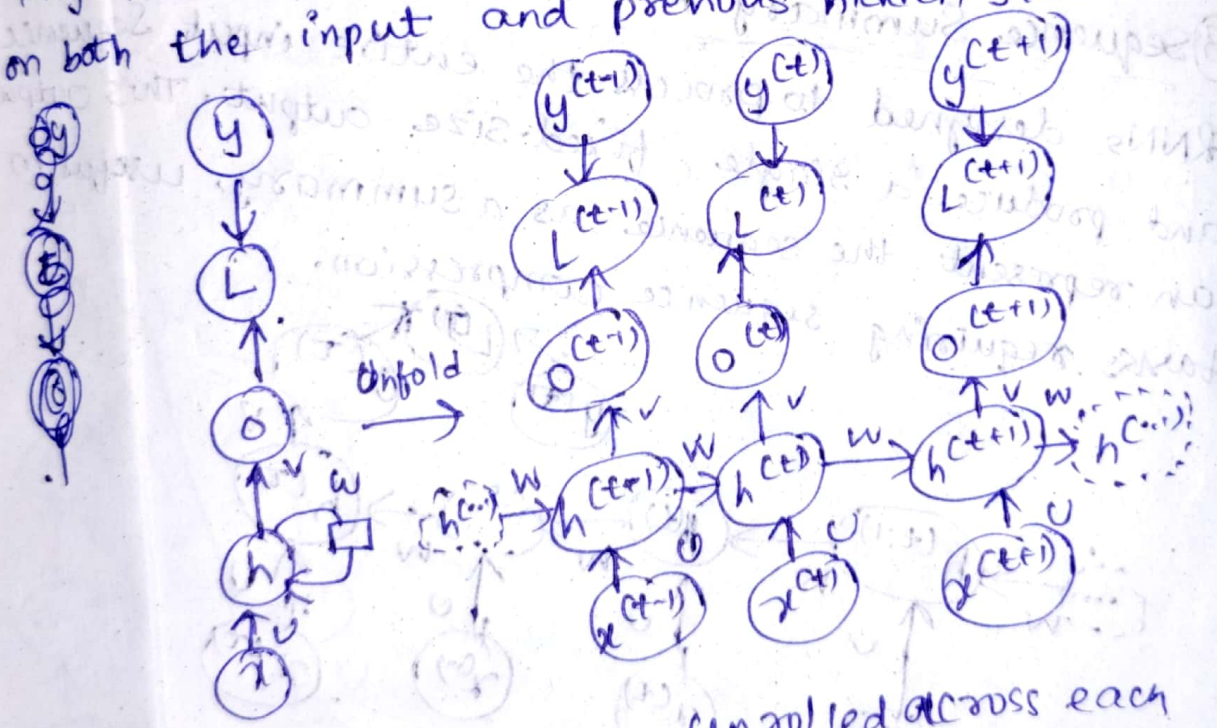
② **Parameter sharing:** The same function f with shared parameters θ is applied at each time step improving efficiency & enabling generalization.

* Recurrent Neural Network *

RNNs are designed to handle sequential data by creating a network with connections that cycle, enabling them to maintain a memory of previous inputs.

. Types of RNN Design patterns:

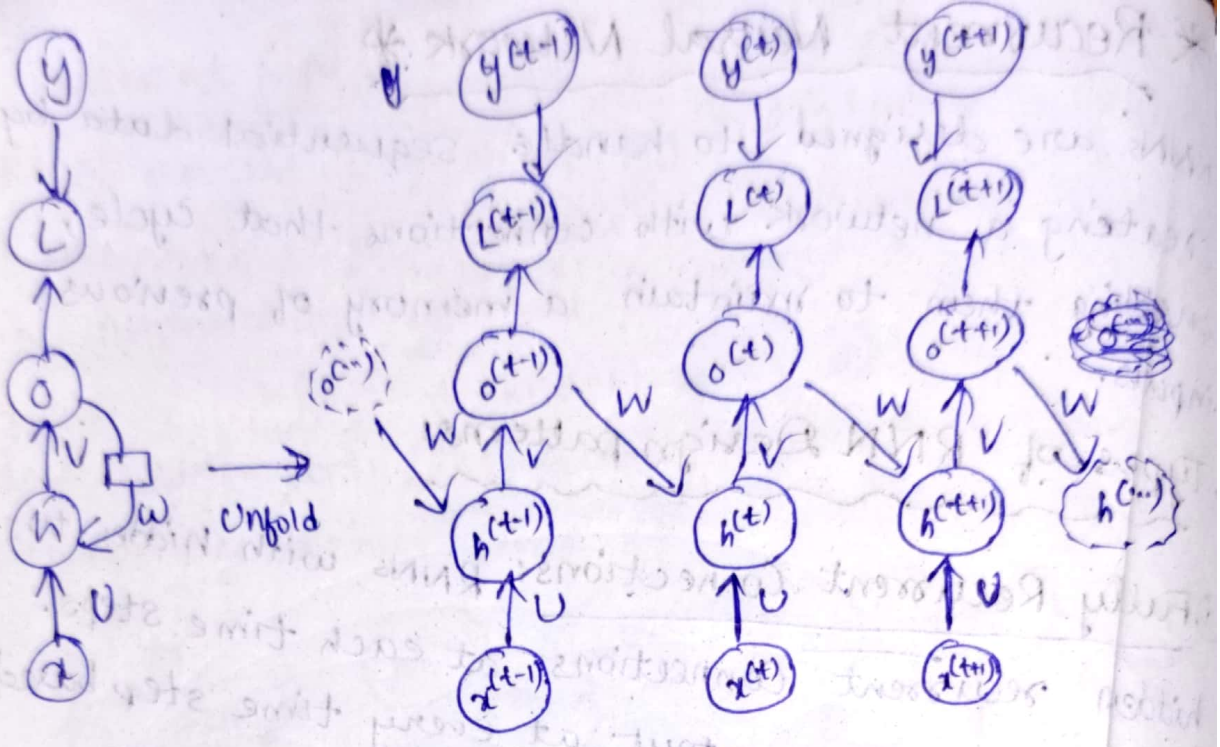
① **Fully Recurrent Connections:** RNNs with hidden-to-hidden recurrent connections at each time steps. They produce an output at every time step based on both the input and previous hidden states.



N/w with direct recurrence

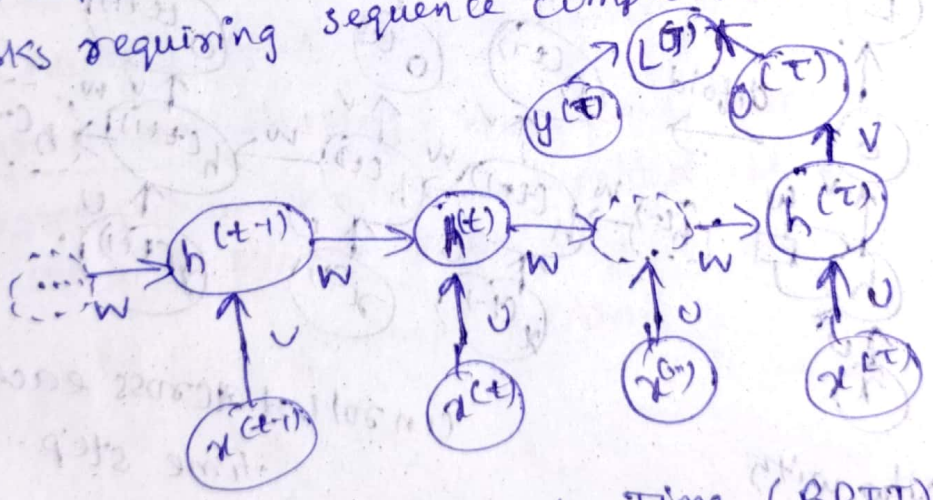
Unrolled across each time step.

② **Output-Driven Recurrence:** RNN with connections from output at one time step to the hidden units of the next lacking hidden-to-hidden connections. This structure may be easier to train but is less expressive due to limited past information retention.



(3) Sequence Summary RNN:

RNNs designed to process the entire input sequence and produce a single, fixed-size output. This output can represent the sequence as a summary, useful in tasks requiring sequence compression.



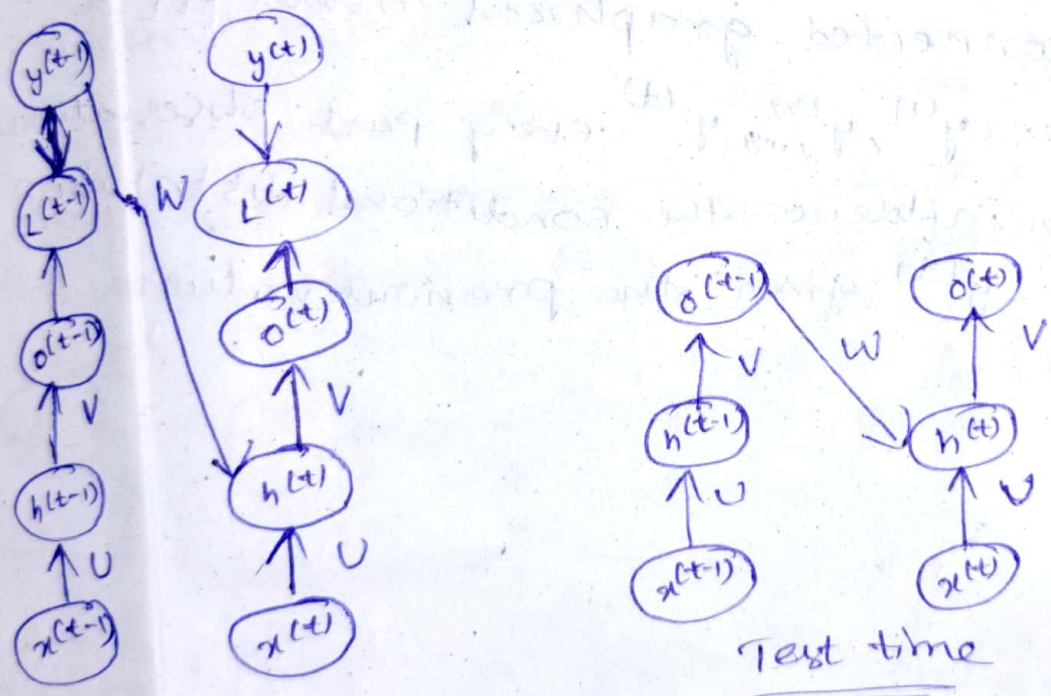
Backpropagation Through Time (BPTT)

Training an RNN through backpropagation involves moving forward through the unrolled computational graph, then moving backward to compute gradients. The BPTT algo calculates these gradients, storing intermediate states for each time step. This method is powerful but computationally memory intensive with complexity of $O(T)$ where $T \rightarrow$ sequence length.

Scanned with OKEN Scanner

⇒ Teacher Forcing in RNN with Output Recurrence:

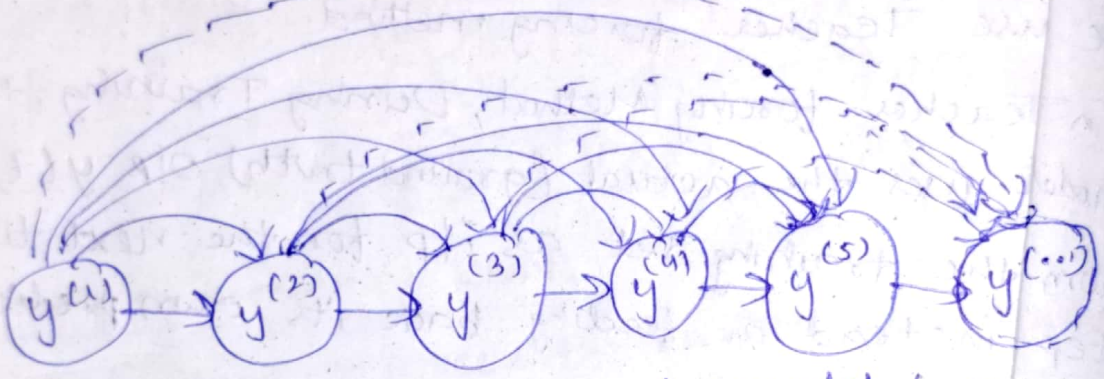
- As we know that the o/p Recurrent networks have recurrent connections from the o/p at one time step to hidden units at the next steps, but lack hidden-to-hidden recurrence.
- Since there are no hidden-to-hidden recurrences we use Teacher forcing method.
- In Teacher-forcing Method, During Training, the model uses the actual (ground truth) o/p $y(t)$ from the training set as i/p for the next time step instead of feeding back its own predictions.



- Teacher forcing is a training technique that is applicable to RNN's that have connections from their output to their hidden states at the next time step.
- At train time we feed correct o/p $y^{(t-1)}$ drawn from trainset as i/p to $h^{(t)}$.
- At test time the model's o/p $o^{(t-1)}$ is feed to $h^{(t)}$

⇒ Recurrent N/w as Directed Graphical Models :—

An RNN can be seen as a directed graphical model, where each prediction depends on the past sequence and is parameterized to capture the joint distribution of sequence.



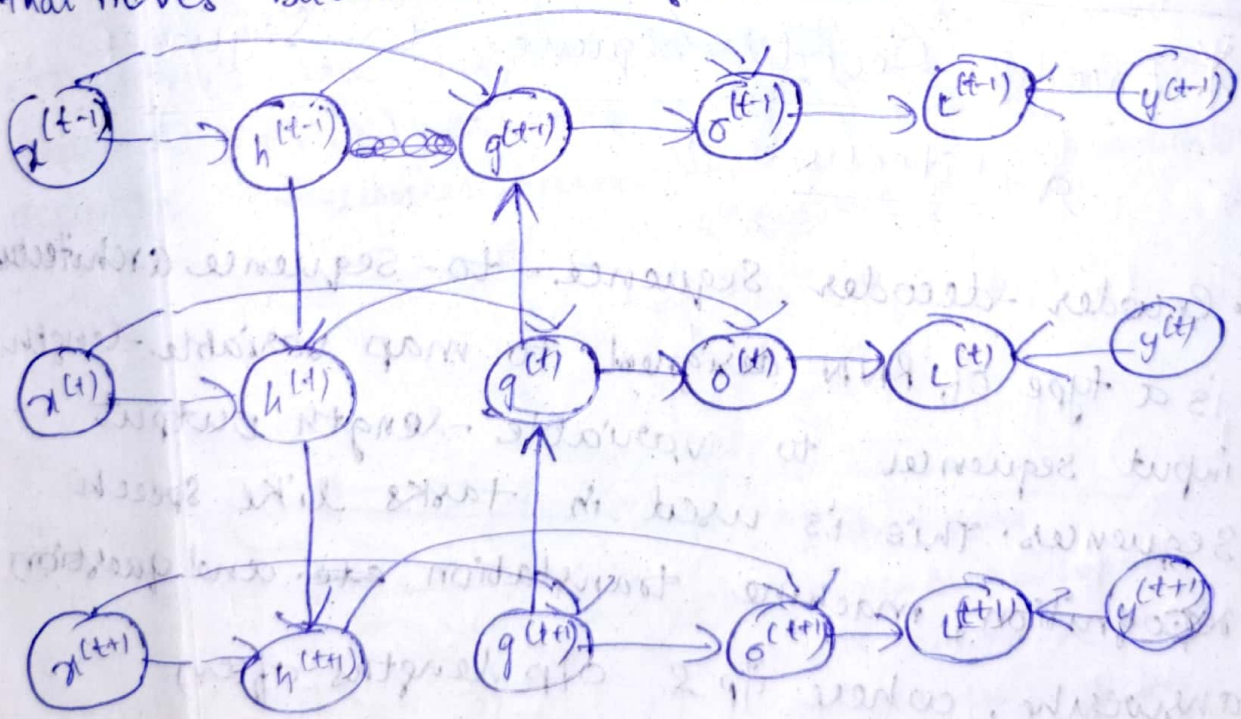
fully connected graphical model for a sequence $y^{(1)}, y^{(2)}, \ldots, y^{(t)}$. every past observation $y^{(i)}$ may influence the conditional distribution of some $y^{(t)}$ given the previous values.

Bidirectional RNN's

- Bidirectional RNN's enhances. Sequence modeling by capturing information from both past and future time steps, making them particularly useful for tasks where future context is needed to interpret the current inputs.
- Unlike traditional RNN's, which only consider past & present inputs, Bidirectional RNN's provides a comprehensive view across the entire i/p sequence, which is essential in Speech recognition and handwriting recognition.

Structure of Bidirectional RNN:

It combines two RNN's one that moves forward through time from the beginning & another that moves backward through time from the end.



The figure represents a bidirectional RNN mapping i/p sequences $x(t)$ to target sequences $y(t)$ with a loss $L(t)$ calculated at each step.

Scanned with OKEN Scanner

- The foward RNN (h-recurrence) propagates information forward, which the backward RNN (g-recurrence) moves backward.
- At each pt t:

  the o/p unit O(t) can summarize relevent past information from h(t) and relevent future information from g(t)

Application:

- Bidirectional RNN's are effective in tasks requiring future context, such as speech Recognition, handwriting recognition & Bio informatics.

---

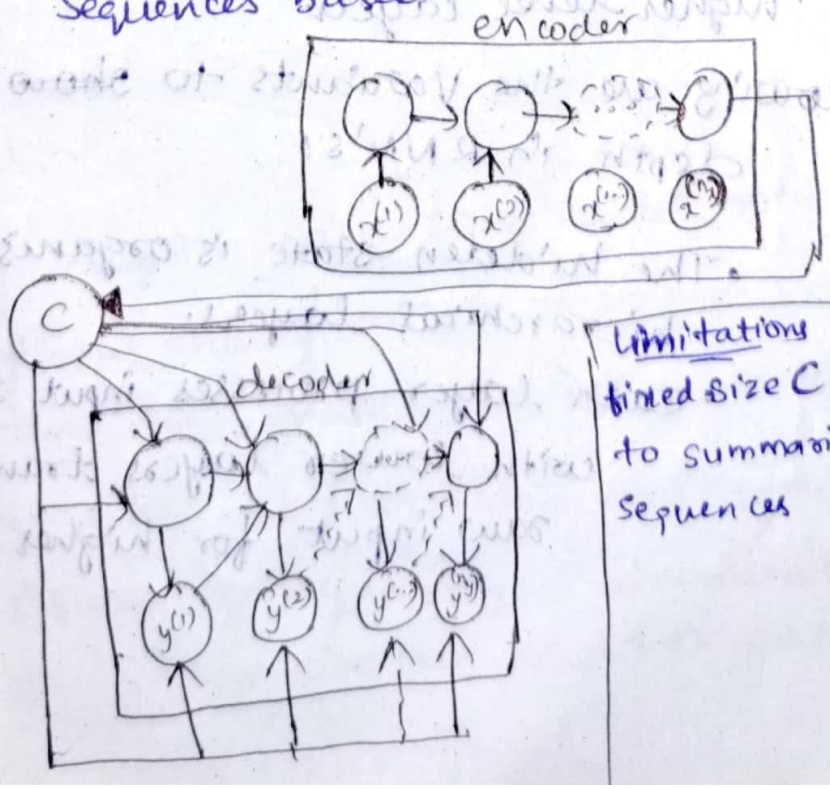
* Encoder-Decoder Sequence-to-Sequence Architectures *

- Encoder-decoder sequence-to-sequence architecture is a type of RNN designed to map variable-length input sequences to variable-length output sequences. This is used in tasks like speech recognition, machine translation and question answering, where i/p & o/p lengths often differ.

Encoder: Encoder Reads the input sequence $X = (x^{(1)}, x^{(2)}, x^{(3)}, \ldots, x^{(n_x)})$ and outputs a fixed-size context vector C, summarizing the input.

Decoder: Uses C to generate the output sequence $Y = (y^{(1)}, y^{(2)}, \ldots, y^{(n_y)})$.

- The lengths of the input (n_x) and output (n_y) sequences can differ.
- The context vector C, is derived from the encoder's final hidden state.
- C can be provided to the decoder as:
  - → initial state
  - → Input at every time step
  - → combination of both
- The RNN's are trained jointly to maximize the likelihood $p(Y|X)$, ensuring the decoder generates accurate sequences based on the encoded context.



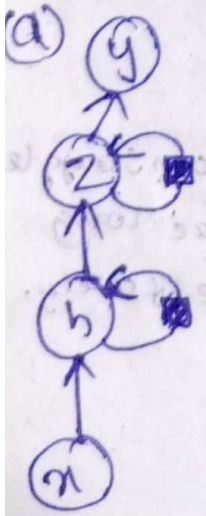
Limitations
fixed size C can struggle to summarize long sequences effectively.

* Deep Recurrent Networks *

- The computation in most RNNs can be decomposed into three blocks of parameters and associated transformations:
  - → from i/p to hidden state
  - → from previous hidden state to the next hidden state.
  - → from hidden state to the output
- Traditional RNNs are shallow in their transformations, using single layer representations.
- By adding depth to this transformations by introducing Multilayer perceptrons (MLP's) to improve performance.
- Introducing multiple layer in the hidden state hierarchy allows lower layers to transform raw input into representations better suited for higher level layers.

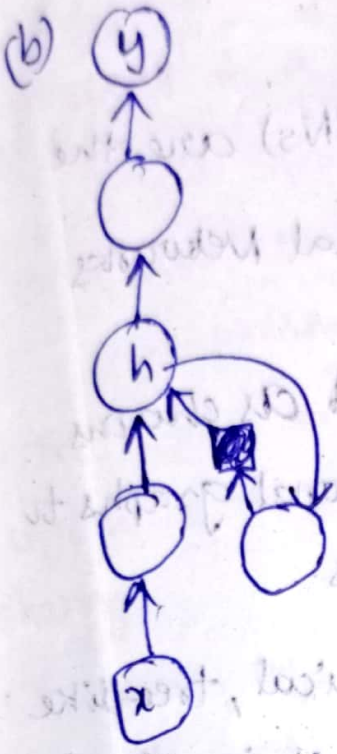
The following are the variants to ~~show~~ incorporate depth in RNN's:

(a)



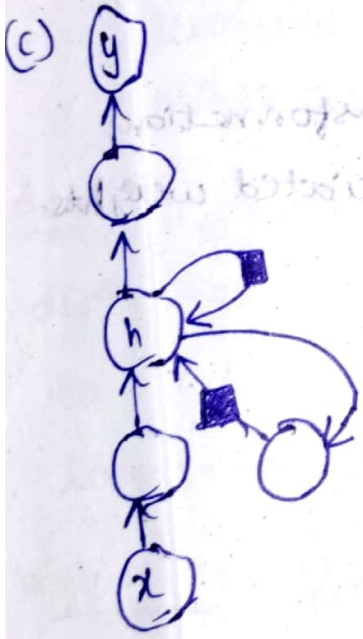
- The hidden state is organised into hierarchical layers.
- each layer processes input sequentially with lower layers transforming raw input for higher layers.

(b)



- This introduces deeper [illegible] in three key RNN transformations:
  - → I/p-to-hidden
  - → hidden-to-hidden
  - → hidden-to-output
- This increases representational capacity but lengthens path blw time steps.

(c)



- This incorporates skip connections in the hidden-to-hidden path to reduce the path length blw time steps
- Skip connections help balance depth and optimization difficulty, improving training efficiency.

Challenges:

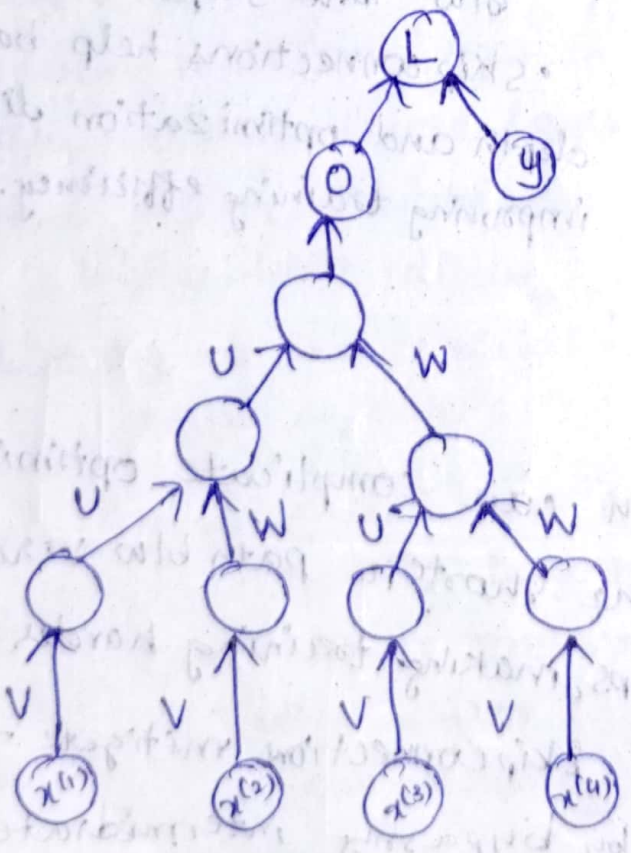
→ Adding depth can complicate optimization, as it lengthens the shortest path blw variables across time steps, making training harder.

→ Solutions like skip connections mitigate this issues by bypassing intermediate layers to maintain shorter paths.

Scanned with OKEN Scanner

* Recursive Neural Networks *

- Recursive Neural Networks (RecNNs) are the generalization of Recurrent Neural Networks (RNNs),
- Unlike RNN, which are structured as chains, RecNNs use tree-like computational graphs to process variable-length sequences.
- RecNNs process data in a hierarchical, tree-like structure rather than the linear chain structure of RNNs.
- each Node in tree computes transformations based on its child nodes and associated weights.



- here, the input sequences $x^{(1)}, x^{(2)}, \cdots x^{(t)}$ is processed in a hierarchical manner, forming a tree structure.
- each subtree's o/p contributes to the final sequence representation.
- The model uses a fixed set of weight matrices U, V, W across all tree nodes, ensuring parameter sharing.
- The final o/p O represents a fixed size vector derived from entire sequence, and a supervised learning target y is associated with this o/p.

Adv: RecNN's reduces the computational depths for sequences from $O(\tau)$ (in RNN's) to $O(\log \tau)$ making better handling long-term dependencies in data.

Applications: Used in NLP & computer vision.

* The challenge of long-term dependencies *

- The challenge of long-term dependencies in RNNs occurs when gradients either vanish or explode when propagated over many time steps. So, this makes difficulty for RNN's to learn dependencies over long sequences.
- Gradients propagated over many stages tends to either diminish (vanish) or grow uncontrollably (explode).
- Vanishing gradients occur when eigenvalues of the weight matrix are less than one, while exploding gradients occur when eigenvalues are greater than one.
- The recurrence relation $h_t = W^t h_0$ leads to eigenvalues being raised to the power t, causing gradients to either decay to zero or explode.
- Even when RNNs can store memories stably, long-term interactions are exponentially weaker than short-term ones.

Learning long-term dependencies becomes extremely slow as gradient signals are dominated by short-term fluctuations.

- Very Deep FeedForward NNs can avoid this problem by carefully scaling weights, as they do not reuse the same weights across multiple time steps.

Scanned with OKEN Scanner

Echo State Networks

An Echo State Network (ESN) is a type of RNN designed to simplify the training process by separating the learning of weights into two parts:

(a) Fixed Recurrent & i/p weights: The recurrent connections & the i/p weights are randomly initialized and remain fixed during training. These weights form the reservior, which serves as rich dynamic system to process i/p sequences.

(b) Trainable o/p weights: only o/p weights are trained using linear regression techniques

Reservoir computing:

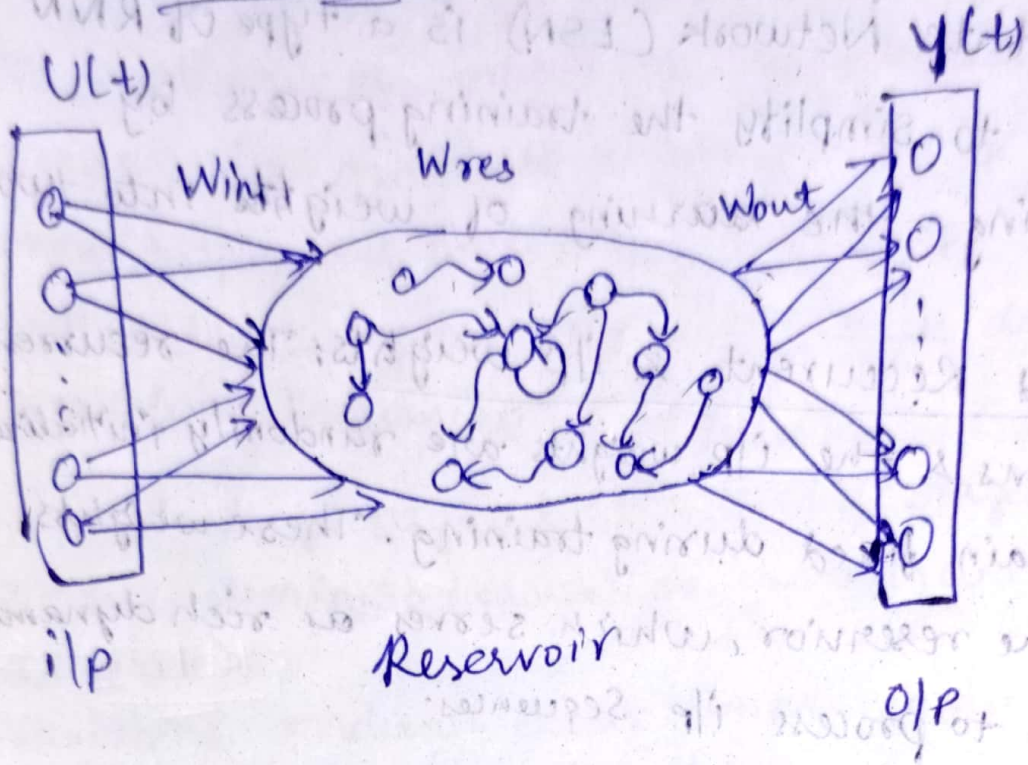
→ The hidden layers (Reservior) acts as a memory system, storing features from the sequence history.

→ The reservoir maps i/p sequences of arbitrary length to a fixed-length representation, enabling the model to handle temporal dependencies efficiently.

* An ESN can be thought of as a Reservoir of temporal features, where the input signal excites the reservoir neurons, creating dynamic responses that represents the history of the i/p. The o/p layer then learns to map these rich dynamic responses to the desired o/ps.

Applications: Time series prediction, signal processing etc

Echo state Nlw



* Leaky units and other Strategies for Multiple Time Scales *

To address long-term dependencies, models can operate at multiple time scales. These scales include fine-grained (handling small details) and coarse-grained (handling long-term dependencies) operations. Strategies include adding skip connections, using leaky units and selectively removing short-term connections.

⇒ Adding Skip Connections through Time:

→ Skip connections adds direct connections between variables from distant past to the present.

→ Skip connections can avoid time delays, reducing gradient vanishing issues by diminishing them exponentially w.r.t delay d.

⇒ Leaky units and a spectrum of different Time Scales:

→ Leaky units are the hidden units with linear self connections, where weights near one retain long-term information, and weights near zero discard it quickly.

→ These units were successfully used in echo state n/ws to handle varying time scales.

⇒ Removing Connections:

→ Actively remove short-term connections and replace them with longer-term ones to encourage long-term processing.

→ The various approaches are:

- Organizing RNN states to flow across longer time scales.
- Using leaky units with diff fixed time scales.
- Implementing explicit, discrete updates at different frequencies for groups of units.

Scanned with OKEN Scanner

* The Long Short-Term Memory and Other Gated RNNs *

- Gated RNNs creates path through time where gradients neither vanish nor explode, allowing for effective learning of long-term dependencies
- Leaky units provided the foundation by accumulating information over time, but gated mechanisms generalized this by making the accumulation dynamic

⇒ LSTM:

→ LSTM networks are an extension of RNN which were introduced by Hochreiter & Schmidhuber (1997) to handle situations where RNNs fail.

- RNN's fail to store information for a longer period of time. i.e, they are incapable of handling "long-term dependencies"
- There is no finer control over which part of context needs to be carried forward & how much of past needs to be "forgotten".
- Other issues with RNN's are exploding and vanishing gradients which occur during the training process of network through back tracking

Hence, LSTM (Long Short term memory) was introduced to overcome this drawbacks.

Scanned with OKEN Scanner

⇒ STRUCTURE OF LSTM:

The basic diff blw the architecture of RNNs and LSTMs in that the hidden layer of LSTM is a gated unit or gated cell.

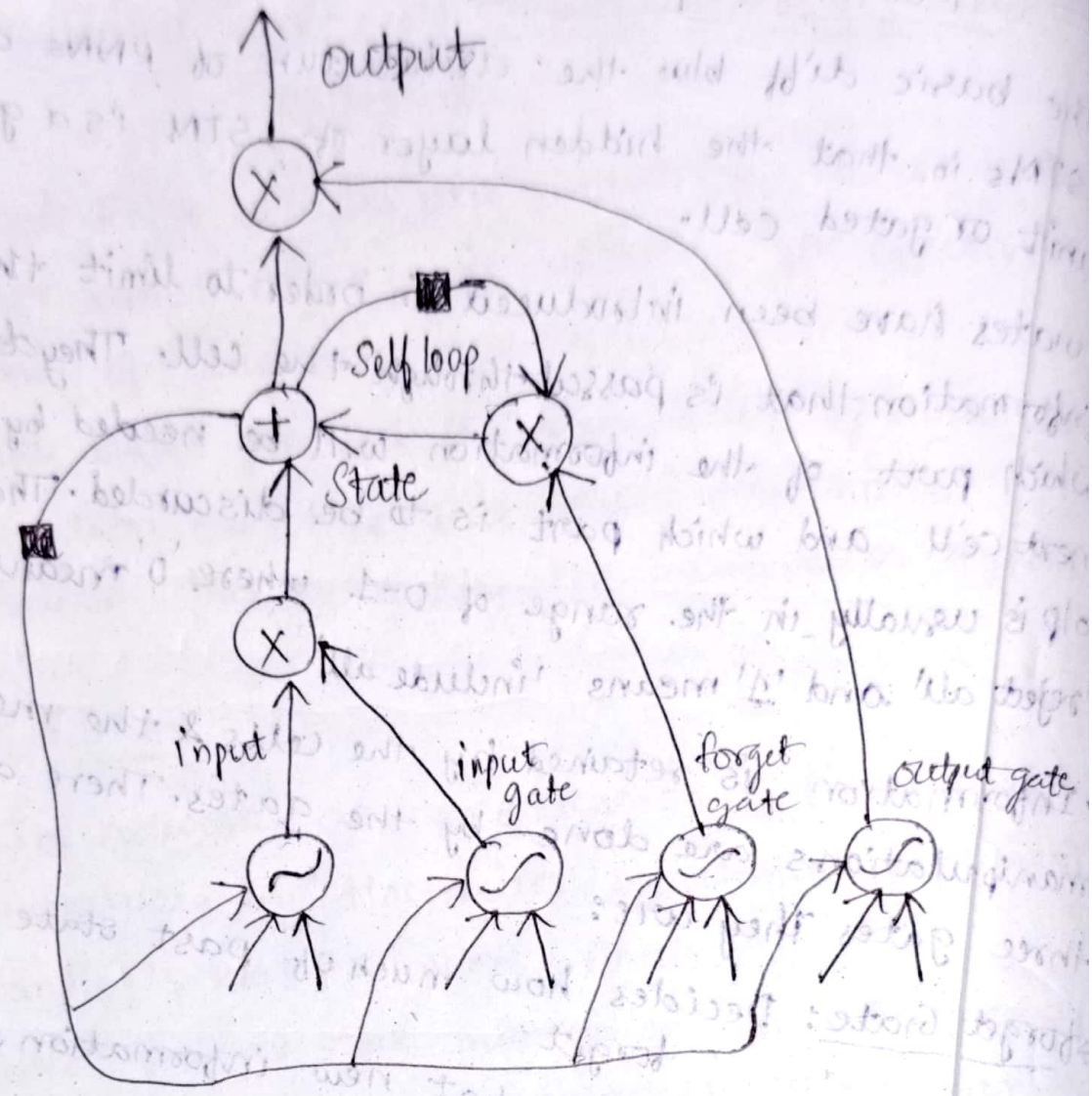
- Gates have been introduced in order to limit the information that is passed through the cell. They determine which part of the information will be needed by the next cell and which part is to be discarded. The o/p is usually in the range of 0-1 where '0' means 'reject all' and '1' means 'include all'.
- Information is retained by the cells & the memory manipulations are done by the gates. There are three gates They are:

→ Forget Gate: Decides how much of past state to forget.

→ Input Gate: Controls what new information should be stored.

→ Output Gate: Determines the o/p of LSTM at each time step.

- These gates ~~are~~ uses Sigmoid activation functions to produce values blw 0 and 1.
- The State unit in LSTM has a self loop with a gated weight, allowing dynamic adjustment of the timescale for integration.
- These are widely used in tasks like handwriting Recognition, Speech recognition, Machine translation, Image captioning. etc



In this figure

- Input Gate determines which i/p feature to be integrated into the State.
- Forget gate controls how much of the past state to retain
- Output Gate regulates the information passed to the next layer or output.
- State units maintains the internal memory of LSTM, featuring a self-loop controlled by forget gate.
- Sigmoid Activation functions are used for gating units and Squashing function (like tanh) for input

And the delay (black square) to indicate a single time step in recurrent connection.

forget gate $f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right)$

input gate $g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right)$

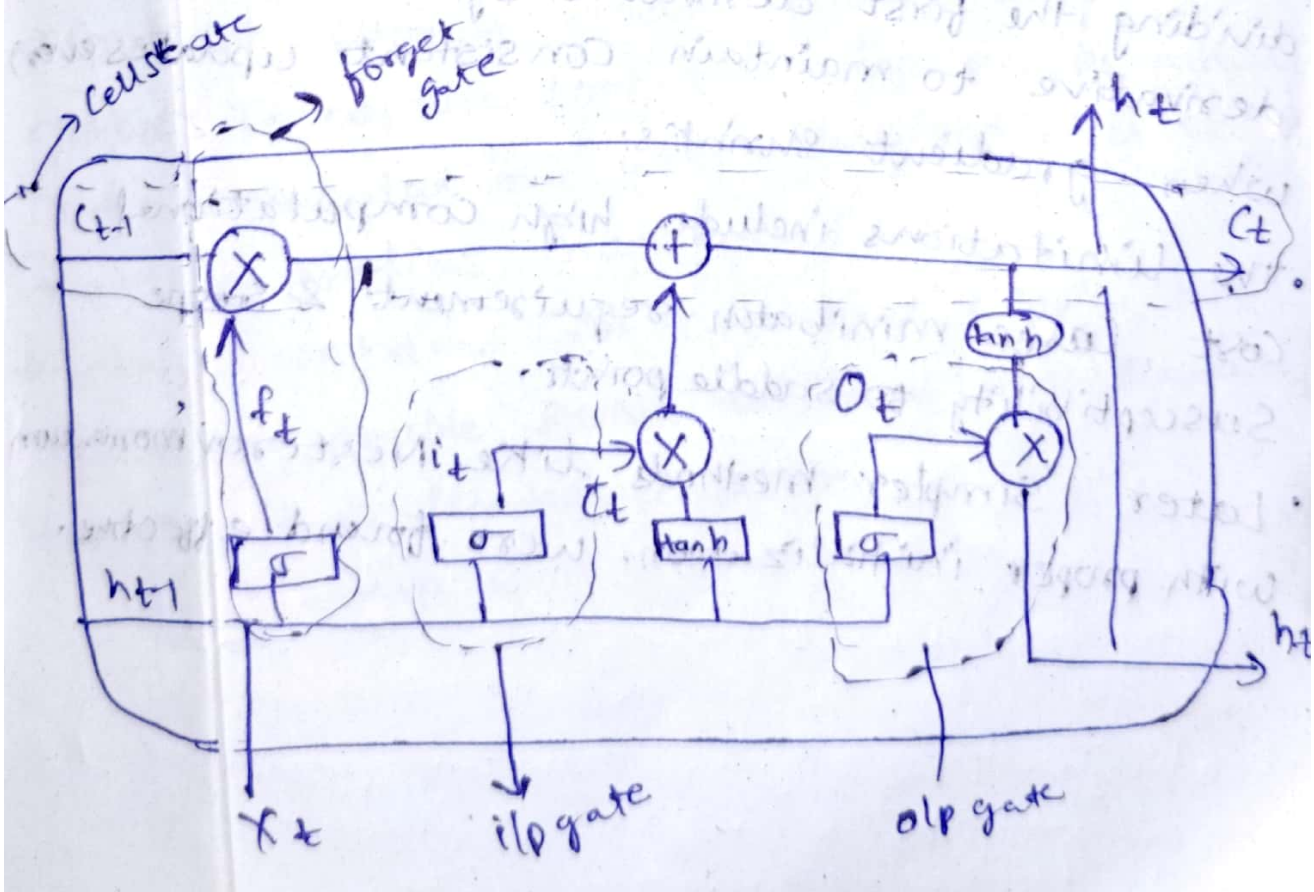
output gate $q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}\right)$

output $h_i^{(t)} = \tanh\left(s_i^{(t)}\right) q_i^{(t)}$

where $s_i^{(t)}$ is ~~self loop~~ state unit

State unit is calculated as:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right)$$



⇒ Other Gated RNN's

→ A simple alternative to LSTM where a single update gate controls both ~~forget~~ forgetting and updating of state.

→ The Reset gate controls which parts of the state are used in computing the next state.

Optimization for Long-Term Dependencies

The various strategies for optimizing RNNs over long time steps, addressing challenges like vanishing and exploding gradients are:

(1) Second-Order Methods:

- Here, ~~Se~~ We use Second-order optimization to address vanishing gradients. These methods approximate dividing the first derivative by the second derivative to maintain consistent updates even when gradient shrinks.
- The limitations include high computational cost, large minibatch requirement & ~~susp~~ susceptibility to saddle points.
- Later simpler methods like Nesterov momentum with proper initialization, were found effective.

② Gradient Clipping:

- Helps manage exploding gradients, which can destabilize training by causing parameters to overshoot optimal values.

→ Element-wise Clipping: Clips individual gradient values.

→ Norm Clipping: Rescales the gradient vector if its norm exceeds a threshold, ensuring updates remain in a descent direction. Norm clipping introduces a bias but works well empirically, particularly for RNNs with steep "cliffs" in the loss landscape.

③ Regularizing ~~for long~~ to encourage Information flow:

→ Gradient clipping doesn't address vanishing gradients. To capture long-term dependencies, pathways in the computational graph are optimized to maintain gradient magnitude.

→ When compared with Gradient clipping, this regularizer enables RNNs to learn longer dependencies though it is less effective than LSTMs, ~~which~~ tasks with abundant data.

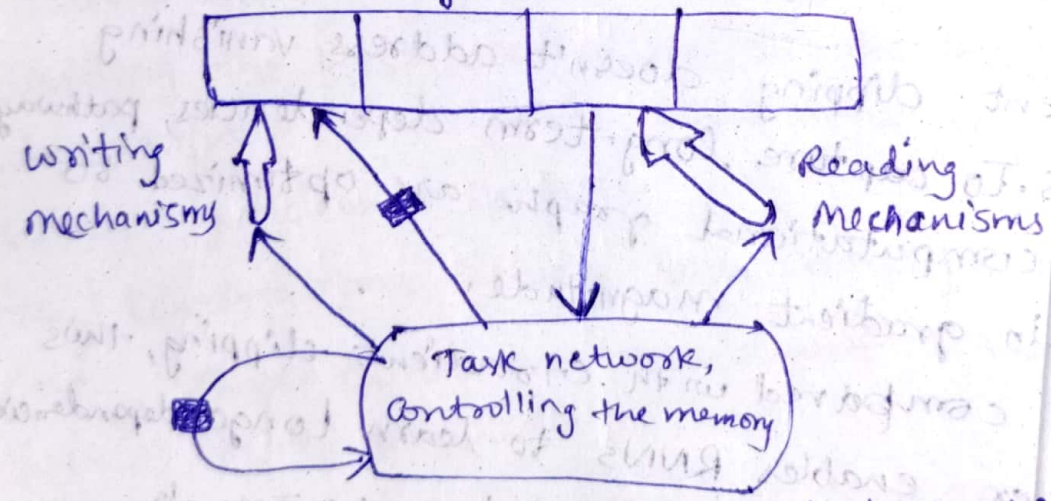
Scanned with OKEN Scanner

* Explicit Memory *

Explicit memory mechanisms in NN are used to enhance their ability to store and manipulate declarative, explicit knowledge (eg specific tasks) rather than just implicit knowledge (eg: Recognizing patterns).

- Neural Networks traditionally struggle with precise fact memorization due to limitations in gradient based learning.
- Explicit memory components, such as memory networks and the Neural Turing Machine (NTM) address these issues by introducing memory cells and addressing mechanisms.

~~Memor~~ Memory cells.



here, A task network (eg: an RNN) is depicted at the bottom, responsible for decision making.

- Memory cells are shown, which can store facts and are separate from the task network.
- Reading and writing mechanisms connect the task network to the memory cells, allowing access to specific memory locations.

Scanned with OKEN Scanner

→ The task network controls memory access by generating "addresses" for reading and writing operations.

→ These mechanisms employ content-based addressing to retrive or update information stored in memory cells.

→ Memory networks uses memory cells for fact storage, initially requiring supervision for their operation.

→ NTM (Neural Turing Machine) extends this concepts by enabling unsupervised reading/writing via content-based soft attention mechanisms

Applications

① Large-scale Deep Learning:

Large scale DL refers to training neural networks with massive datasets and significant computational resources to solve complex problems. These includes:

→ Distributed computing: Large scale DL leverages distributed systems (like Google's Tensorflow) to train models across multiple GPUs/TPUs

→ Data Optimization: Large Datasets allows the model to learn more intricate patterns, improving performance on tasks such as img recognition or language modeling

→ Applications:
- Autonomous sys for self driving cars

* Computer Vision *

- Computer vision simulates or extends human visual capabilities, such as recognizing objects & faces
- Deep learning has revolutionized CV, enabling systems to analyze and interpret visual data. Key contributions include:

→ Object Detection: Identifying objects in image or videos (eg: YOLO, Faster R-CNN)

→ Image Classification: Categorizing images into predefined classes (eg: ResNet, VGG)

→ Semantic Segmentation: Partitioning an image into region based on context (eg: U-Net, Deeplab)

→ Preprocessing in CV: like Rescaling, Dataset Augmentation which enhances the training dataset by adding variations, reducing overfitting

→ Contrast Normalization: (GCN)

- Global contrast Normalization adjusts image contrast by subtracting the mean intensity and rescaling pixel standard deviation to a constant
- Local contrast Normalization (LCN) focuses on normalizing contrast in small regions, highlighting local features like edges & textures

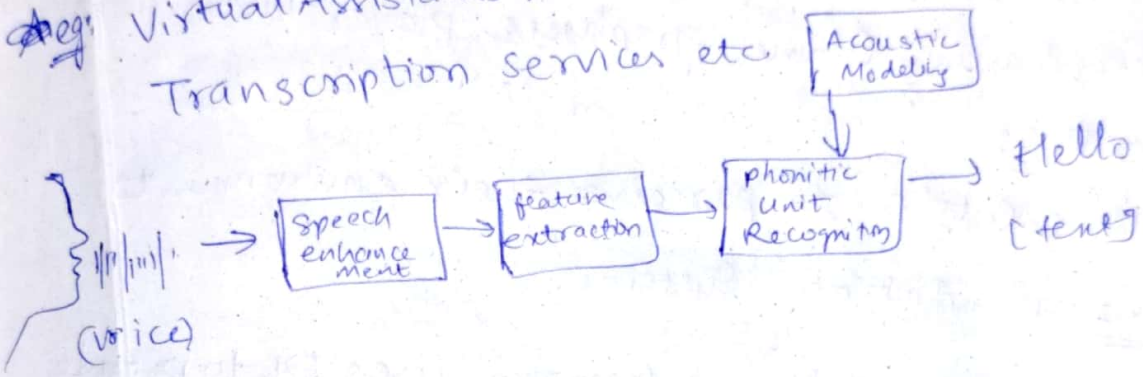
* Speech Recognition *

DL enables systems to convert spoken language into text with high accuracy. Key technologies includes:

→ End-to-End Models: Models like DeepSpeech simplify the process by directly mapping audio to text.

→ RNNs and Transformers: Handle sequential data effectively capturing the temporal patterns in speech.

eg: Virtual Assistants like Alexa, Siri etc
Transcription services etc



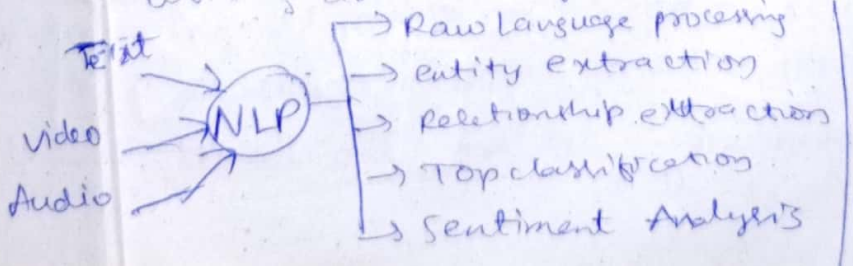
* Natural Language Processing (NLP) *

Deep learning has enhanced NLP by allowing models to understand, generate and manipulate human language. Major advancement includes:

→ Language Models: pre-trained Models like GPT, BERT etc
Enables tasks like summarization, translation and question answering.

→ Sentiment Analysis: Determining emotions or opinions expressed in text.

→ Text Generation: Generating coherent Texts, such as writing articles or generating code snippets.



eg: Chatbots

Other Applications

→ **Health care:**
- predictive analytics for patient care.
- Drug discovery and genomics

→ **Finance:**
- fraud detection in transactions
- Stock price predictions using time-series data

→ **Gaming:**
- AI players like AlphaGo defeating human champions
- enhancing Game graphics & player interactions.

→ **Robotics:**
enabling Robots to perceive their environment

→ **Recommendation Systems:**
- Netflix, YouTube & Amazon uses DL to provide personalized content recommendations

→ **energy & climate:**
- Predicting weather patterns ~~& optimizing~~

RNN :-

→ Recurrent neural N/w is a type of neural N/w where the O/p from the previous steps as fed as input to the current step in tradition neural N/ws.

→ All the inputs & O/ps are independent of each other.

→ The main important features of RNN is Hidden state, which remembers some information b/w a sequence

→ It is used to handle sequential data

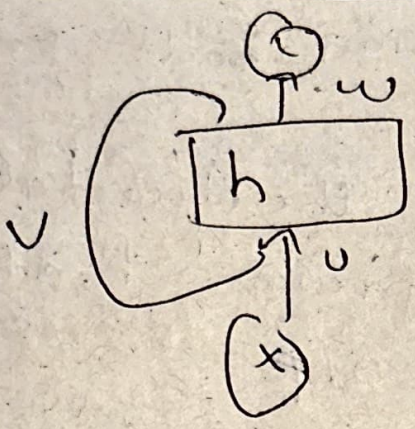
Key Components of RNN

There are mainly 2 components of RNN's

a) Recurrent neurons :- The fundamental processing unit in RNN is Recurrent unit.

→ They hold the hidden unit that maintains the infor about the previous input in sequence.

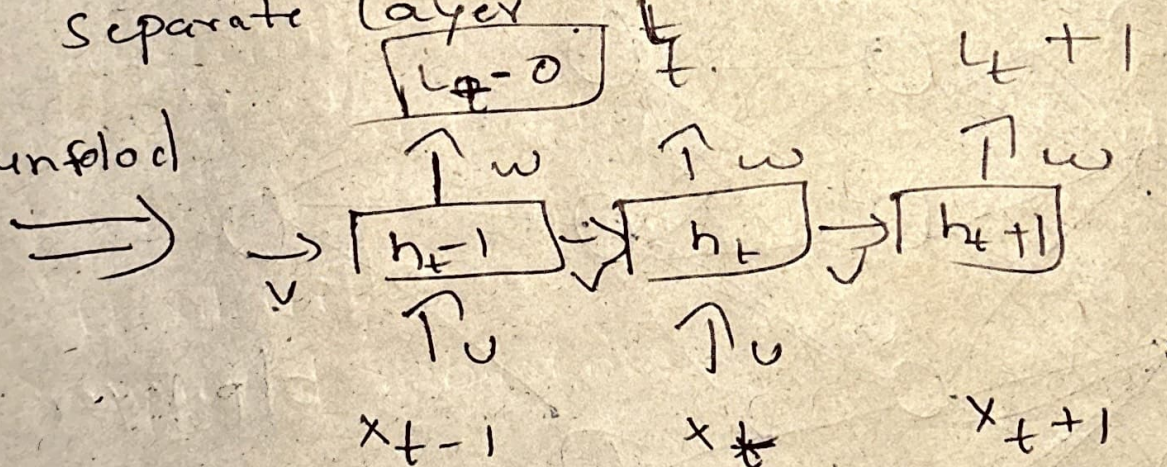
→ Recurrent units can remember information from previous steps. allowing them to caputure dependenc



b) RNN unfolding :-

RNN unfloding or unrolling is the process of expanding the recurrent structure over time steps.

→ During unfolding, each step of the sequence are represented as a separate layer



How RNN works

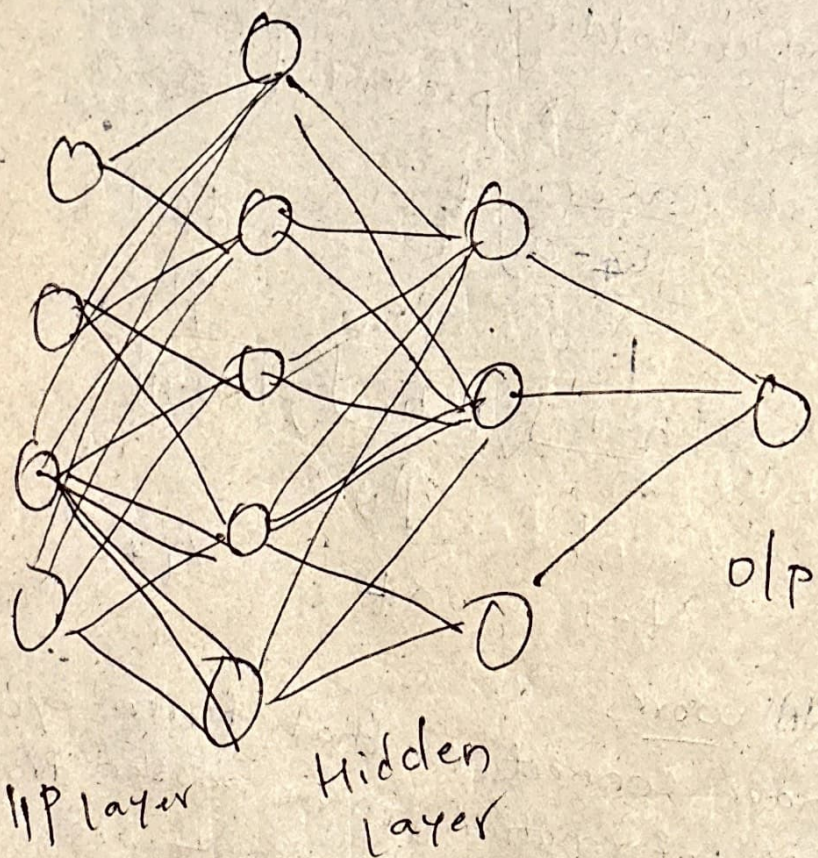
→ RNN have connections that form cycles, allowing information to be passed from o/p layer to its input in current step.

→ This recurrent connection is facilitated with hidden state, which acts as a form of memory

that stores information about previous i/p in the sequence.

→ At each time step, the Nlw takes the current i/p & previous hidden state to produce o/p.

→ RNN uses the same set of weights for each time steps, which reduces no. of parameters.



→ RNN performs well on short sequences, but struggles to capture long-range dependencies.

Basic structure

Input layer :- Receives the sequence data

Hidden layers :- processes input & maintains information from previous time steps through recurrent connections

O/p Layer :- Generates predictions based on the current hidden state.

~~Types~~ ~~RNN architecture~~

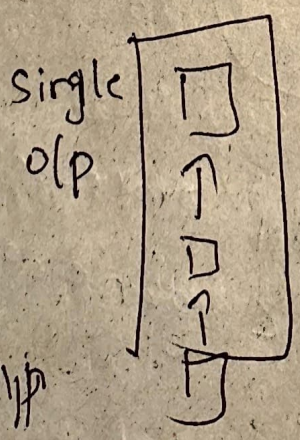
Types of RNN :-

→ There are four types of RNNs based on the number of I/p & O/p in the N/w

i) one-to-one RNN :-

This is the simplest type of N/w architecture where there is single I/p and a single o/p.

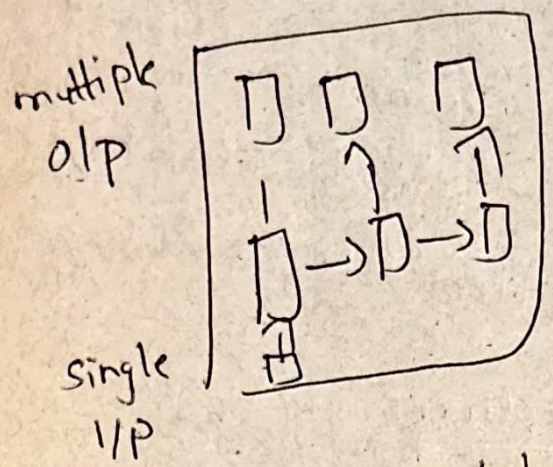
→ It is used for straigh forward tasks like binary classification.



ii) One-to many :-

In one-to-many RNN the NN processes a single I/p to produce multiple o/p.

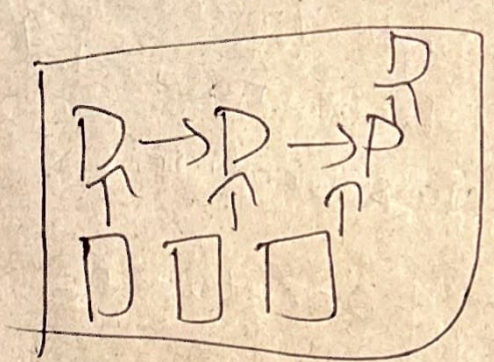
→ This is useful in tasks where one input triggers a sequence of o/ps.



iii) many-to-one RNN. NN

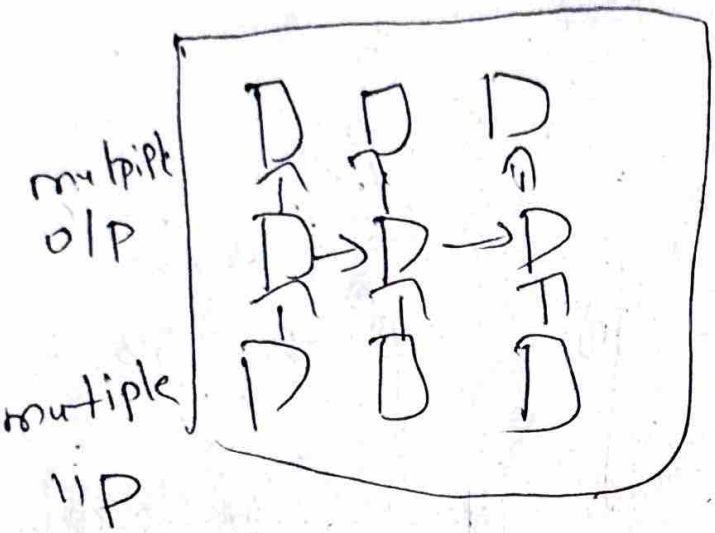
In many-to one RNN, receives a sequence of I/p to generate a single o/p.

→ This type is useful when the overall context of the input sequence is needed to make one o/p.



iv) many-to-many :-

→ many-to many RNN processes a sequence of inputs & generates a sequence of o/ps used in language translation.



Application :-

1) NLP
2) speech recognition
3) image & video processing
4) Text summarization
5) stock market forecasting.

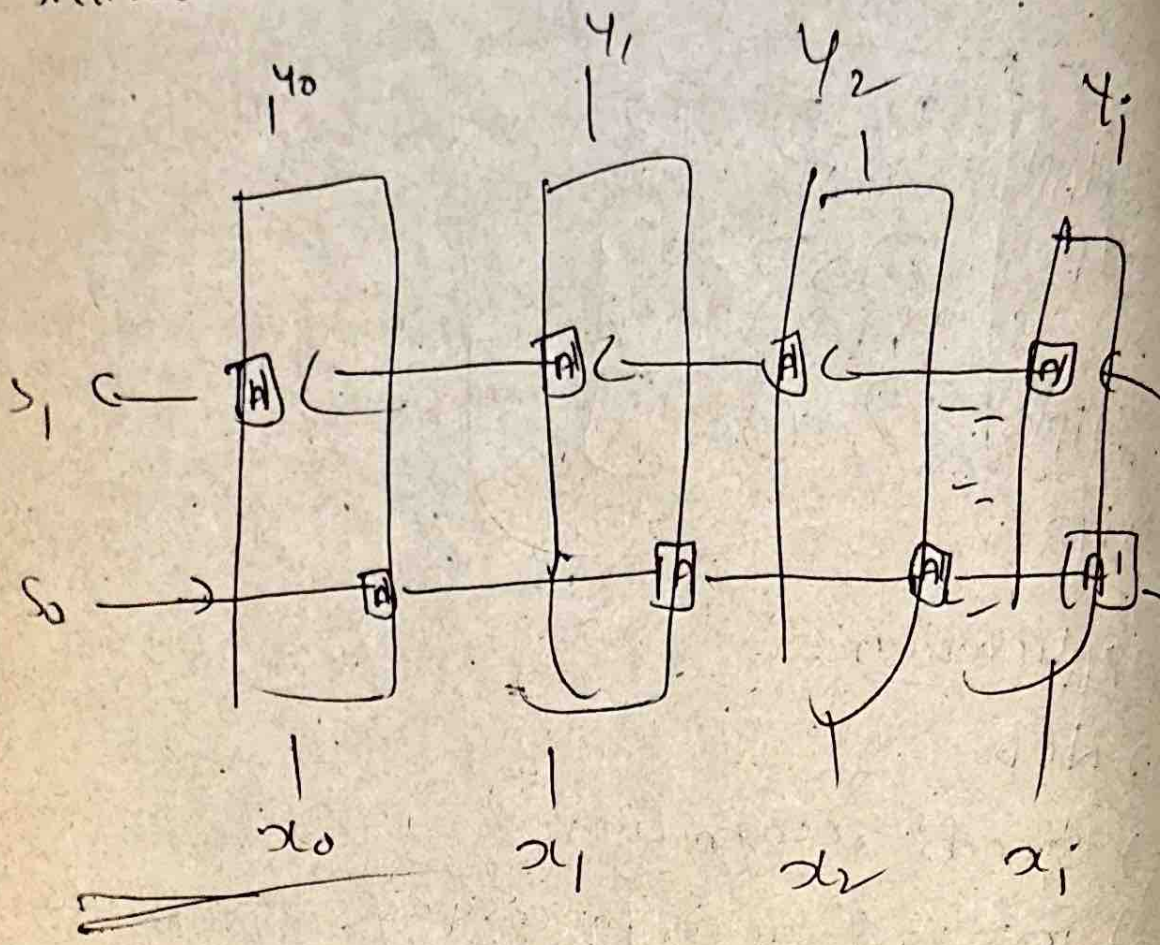
Advantages

-> processes sequence of any length
-> parameter sharing.
-> models time-dependent data

Disadvantages :-

-> Difficulty with long sequences
-> slow & complex training
-> limited memory
-> They suffer from vanishing gradient problem, making it difficult to learn long-term dependencies.

Bidirectional RNN



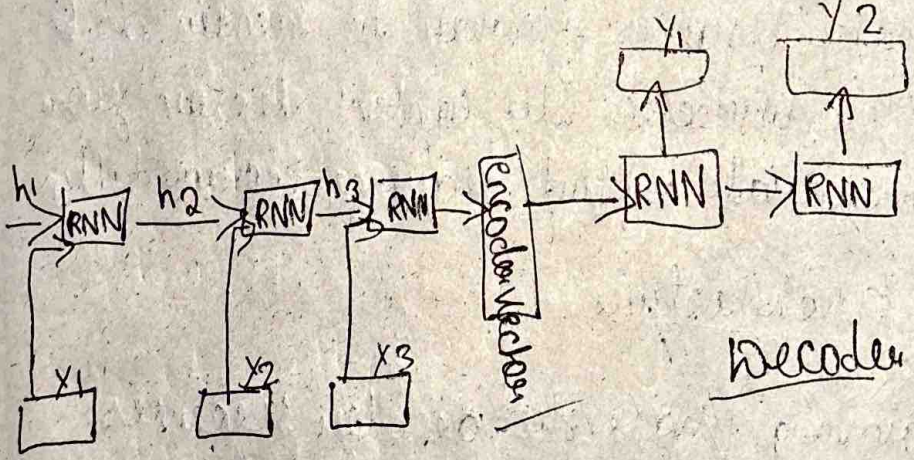
Encoder-Decoder Sequence-to Sequence Architecture

- The encoder-decoder sequence-to-sequence (seq2seq) architecture is a machine learning architecture designed for tasks involving sequential data.
- It takes an input sequence, processes it, and generates an output sequence.
- The architecture consists of two fundamental components: an encoder & a decoder.
- The encoder processes the input sequence and transforms it into a fixed-size hidden representation.
- The decoder uses the hidden representation to generate output sequences.

- The encoder-decoder structure allows them to handle input & output sequences of different lengths making them capable to handle sequential data.
- The model is trained to maximize the likelihood of the correct output sequence given the input sequence.
- Commonly used in tasks involving NLP, speech recognition, Machine translation or question answering
- Where the input and output sequences in the training set are usually not of the same length (although their lengths might be related)
- Imagine we have an input sentence:

  "The sky is"
- The correct output word (what that model should predict) is:

  "blue"



Encoder Block

The main purpose of the encoder block is the process the input sequence & capture information in a fixed-size context vector.

Encoder

1. The input sequence is fed into the encoder.
2. The encoder processes each element of the input sequence using neural Network (or transformer architecture.
3. Throughout this process, the encoder keeps an internal state, and the ultimate hidden state functions as the context vector that encapsulates a compressed representation of the entire input sequence.
4. This context vector captures the semantic meaning & important information of the input sequence. //

Decoder Block

- The decoder block is similar to encoder block
- The decoder processes the context vector from encoder to generate output sequence incrementally

Decoder Architecture

- In the training phase, the decoder receives both the context vector & the desired target output sequence (ground truth)
- During inference, the decoder relies on its own previously generated outputs as inputs for subsequent steps.

Applications

1) machine translation
2) Text summarization
3) chatbots
4) Image Classification

Limitations :-

The bottleneck problem :-

→ The main limitation of basic seq-to-seq architecture is context vector.
→ No matter how long the input sequence is, its information must be compressed into single fixed size vector.
→ for very long sequences, this causes an information bottleneck, leading to loss of information

LSTM (long short term memory) :-

→ LSTM is a specialized type of RNN that excels at learning from sequential data by using a system of gates to manage memory cells.
→ It solves the vanishing gradient problem of standard RNNs, allowing

it to remember & proocess long-term dependencies of data like text, speech or time series.

→ Key components are forget, input & output gates, which regulates the information flow.

LSTm architecture

LSTm architecture involves the memory cell which is controlled by three gate

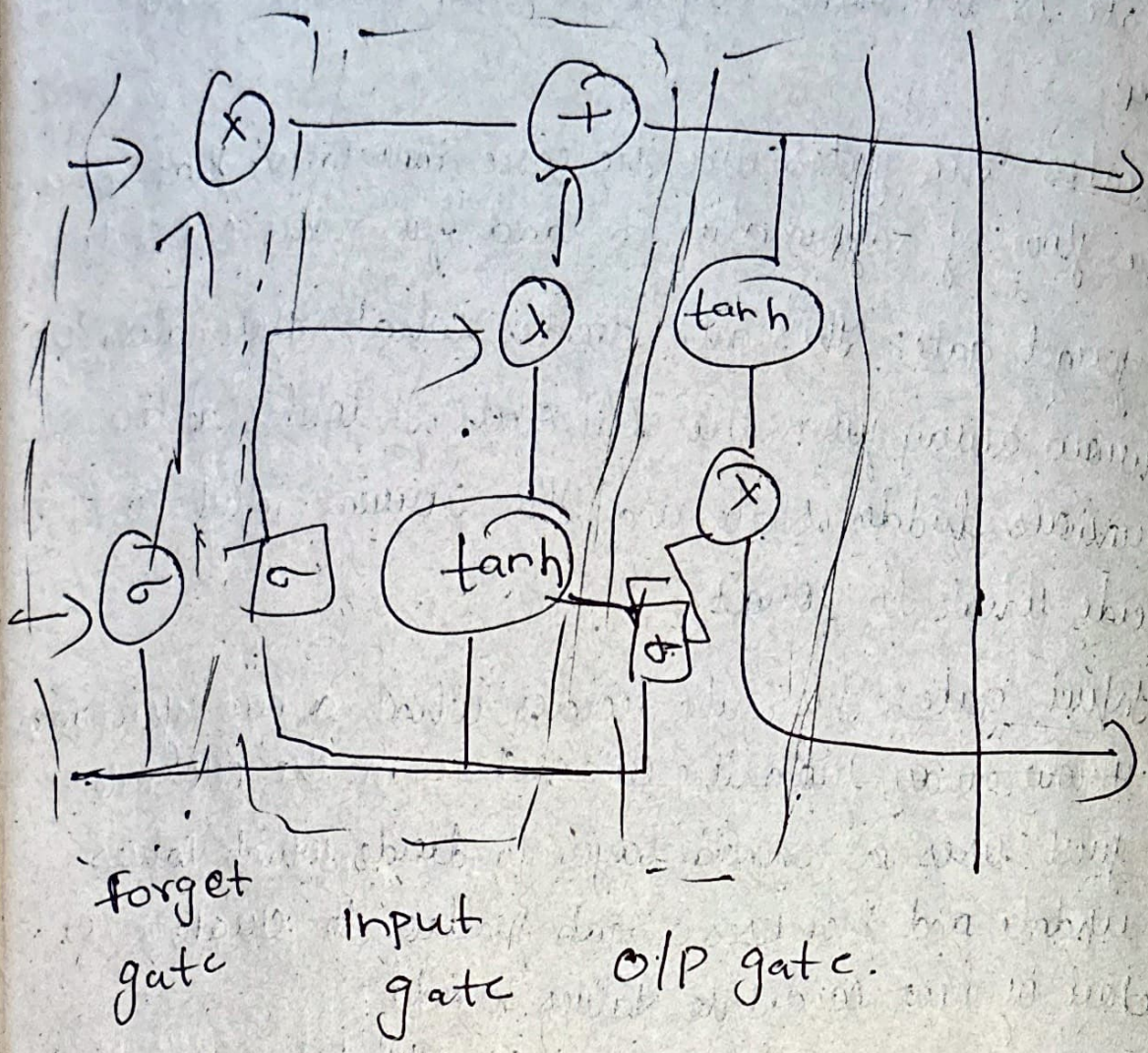
i) input gate
ii) o/p gate
iii) forget gate.

forget gate :- decides what information to discard from the memory cell.

input gate :- decides what new information to store in memory cell

o/p gate :- controls what information is o/p from the memory cell.

This allows LSTm nlw to selectively retain or discard information, which allows them to learn long-term dependencies.



⊗ point-wise multiplication

⊕ point-wise addition

tanh pointwise tanh

σ sigmoid activation

tanh Tanh activated NN.

How it works:-

memory cell and Hidden State :- LSTMs have a "cell state" & Hidden State". The cell state is like a conveyor belt that runs straight down the

entire chain, carrying information, while the hidden state is the ~~same~~ output for the current time step.

- Gates: The gates are the core innovation that control the flow of information to and from the cell state.
  - forget gate: This gate decides what information to throw away from the cell state. It looks at the previous hidden state and the current input to decide what to forget.
  - Input gate: This gate decides what a new information to ~~throw away from the cell s~~ store in the cell state. It just uses a sigmoid layer to decide which values to update and then uses a tanh function to create a vector of new candidate values to add.
  - output gate: This gate decides ~~which~~ what part of the current cell state to output. It just runs a sigmoid layer to decide which parts of the cell state to output & then multiplies that with a filtered version of the cell state.

Applications

1) Natural language processing
2) Speech Recognition
3) Time series forecasting
4) Other areas

Advantages

- Handle long-term dependencies: unlike standard RNNs,

LSTMs are specifically designed to maintain information over long sequences without the vanishing gradient problem

- selectively remembers information: the gating mechanism allows the network to forget irrelevant information and focus on what's important.

→ Explict memory

-> Explict memory refers to a approach that uses separate, external memory components for storing & retreiveing specific information in structured way.

-> The explicit memory models uses external memory banks to store facts, data or specific data points for more precise & interpretable recall

-> This can be achieved using architectures like Neural tuning machine or differentiable neural computer, which explicity write to & reads from external memory enhancing performance

Recursive Neural N/w :-

RNN is a type of Neural N/w designed to handle hierarchical structures, making them suitable for stasks involving tree-like or nested data.

→ RNN model applies same set of weights recursively over structured input to produce structured o/p.

→ RNN operates on hierarchical, tree like structure.

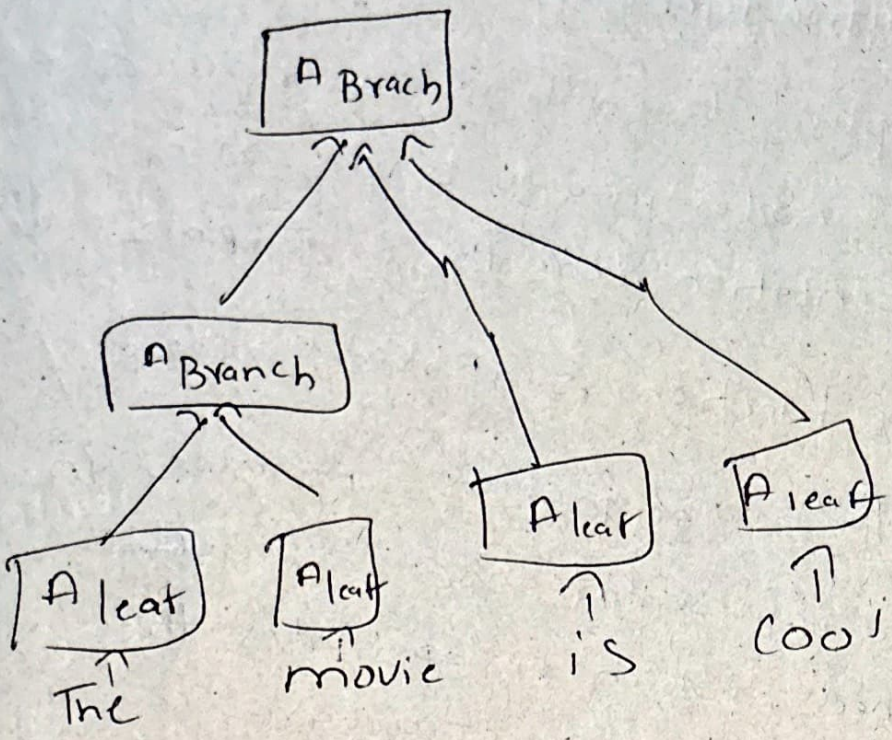
→ The primary purpose of RNN is to handle hierarchy data.

→ It uses a single, fixed set of weights to process data at different levels of hierarchy & merging child representations into parent representations until it reaches the root of the structure.

Architecture & Structure :-

The architecture is defined by the I/P tree structure.

→ The N/w is built upwards from the leaves to the root



Input Data :- The input is not a flat array but a tree (Ex:- A prase tree).

Nodes :- Each node in the tree represents data points.

leaves :- These are the initial input vectors.

Interal nodes :- These are calculted by combining the representations of their children using neural N/w function.

Working :-

-) Bottom-up processing :- The N/w starts at the lowest level

Combination function :- A fixed function is applied to combine the representation of child nodes into a single, rich representation for their parents node.

→ This function is applied repeatedly.

final o/p :- The process culminates in a single representation at the root node, which encapsulates the meaning of entire structure.

mathematical formulation :-

for parent node p with children $c_1, c_2, \ldots c_k$, the parent's representation h_p is calculated as

$$h_p = f\left(W \cdot [h_{c_1}; h_{c_2}; \ldots h_{c_k}] + b\right)$$

h_{c_i} = vector representation of child i

W & b are the shared weights & bias.

f is non-linear activation function.

Advantages :-

→ Handles structures :- Naturally models hierarchical relationships.

parameter efficiency :- uses a small, fixed set of shared parameters regardless of tree size.

→ learns how meaning is built from sub-parts.

Limitations :-

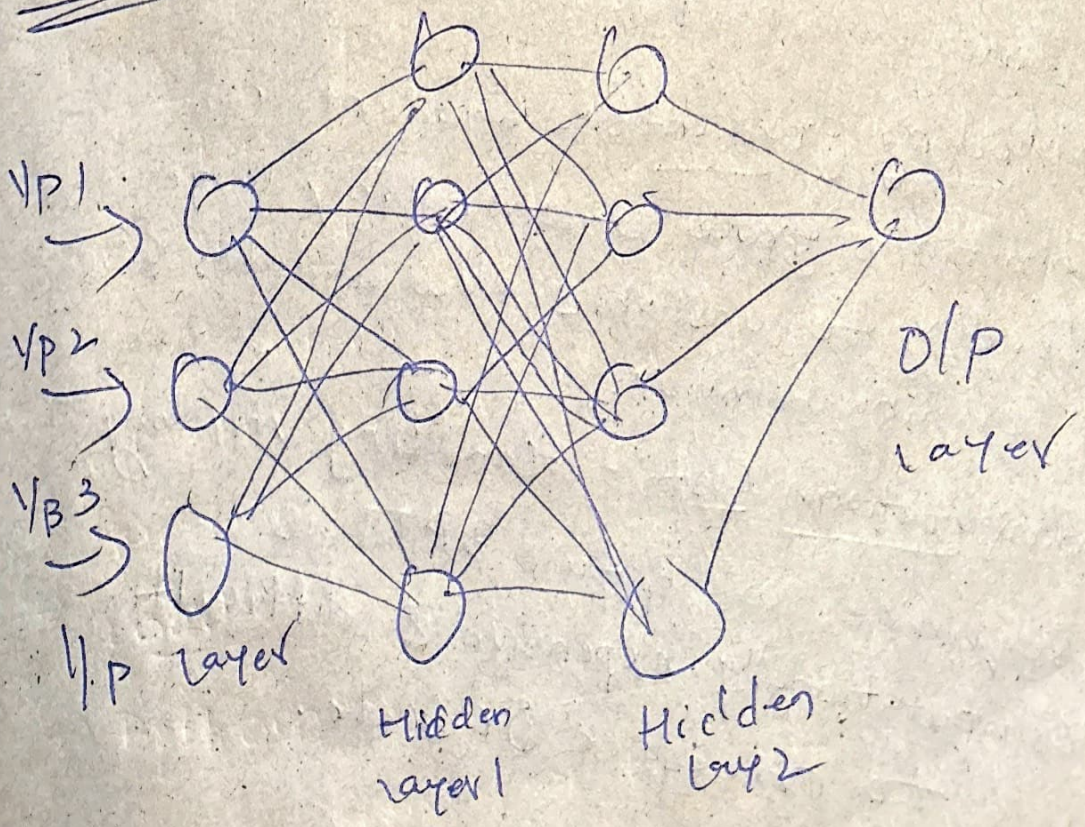
Requires Defined structure :- It needs predefined i/p tree structure

→ computational cost :- Can be complex to implement efficiently

Applications

→ NLP.

→ computer vision



Challenges & optimization of long-term dependencies:-

The primary challenge in handling long-term dependencies in sequential data is the vanishing & exploding gradient problem.

-> The core difficulty arises during the training the process using backpropagation through time, where gradients are propagated through many time steps.

Vanishing gradients:- Gradients often shrink as they propagate backward through time.

-> This makes it difficult for the model to learn & adjust weights for events.

-> Consequently the model fails to capture long-term pattern

Exploding gradients:- Gradients can sometimes grow uncontrollably large leading to unstable n/w updates & learning behavior.

Computational cost:- Handling very long sequences with full BPTT is computationally expensive & difficult to measure or scale.

optimization :-

several architectural & algorithmic solutions have been developed to optimize the learning of long-term dependencies.

1) Gated architecture :-

- Long short-term memory (LSTM) :- This is a specialized RNN architecture that uses memory cells & distinct gates to control the flow of information.
  -> The forget gate selectively removes irrelevant info, allowing the N/w to retain important info over long sequences & avoid vanishing gradient problem.
- Gated Recurrent units :- A simplified version of LSTM that combines I/p and forget gates into a single update gate.
  -> It is more computationally efficient & performs similarly to LSTM
- Bidirectional RNNs (BRNNs) :- These N/w processes data in both forward & backward

directions, allowing the o/p at any point to benefit from both past & future context.

=

Echo state n/w :-

-> Echo state N/w is a type of RNN used for sequence modeling

-> It is efficient to train model because its internal recurrent connections are fixed & randomly initialized.

Leaky units :-

-> A leaky units is an activation function that allows to give small, non-zero o/p even when the i/p is -ve.

Unit-V

1) Performance Metrics:

In DL, practical methodology means how we evaluate models, in real-world situations. performance metrics are important part of this methodology because they tell us how well a model is learning, how accurate it is, and whether it is suitable for deployment. Different tasks need different metrics, and choosing the right one helps us judge the true performance of a model.

classification problems:

→ For classification problems, accuracy is the simplest metric. It tells us the percentage of correctly predicted samples. But accuracy alone is not enough, especially when the data is imbalanced. Therefore, we use metrics like precision, recall and F1 score. precision shows how many predicted positives were actually correct, recall shows how many actual positives were correctly found by the model, F1 score balances both.

→ This metric helps us understand the model's behaviour deeply, especially in sensitive applications like medical diagonsis or fraud detection.

Multi-class and Multi-label problems:

→ For this we use metrics like macro, micro and weighted averages. These metrics ensure we evaluate all classes fairly, even if some categories have fewer samples. confusion matrices are also used, as they visually show where the model is making mistake.

Regression Tasks:

→ We use metrics such as Mean squared Error (MSE), Mean Absolute Error (MAE), and R^2-score.

- MSE and MAE measure how far the model's predictions are from actual values. MSE punishes large error heavily, while MAE gives equal weight to all errors. R^2- score shows how well the model explains the variability of the target variable.
- These metrics help us understand model accuracy in numerical predicition tasks.

~~LOO~~ Loss curves:

In practical Methodology, we also check loss curves, which show the model is learning during training. By watching training and validation loss, we can detect problems like overfitting or underfitting.

Default Baseline Models

→ In DL, a default baseline model is the simplest model we build first before trying the more complex models. The purpose of a baseline is not to get the highest accuracy but to provide a reference point.
→ It helps us understand whether our advanced model is actually learning something useful or not.
→ If a complex dl model cannot beat a simple baseline, then it means there is a problem with the data features or model design.

1) The Default Baseline Models are of 4 types, they are:

i) Logistic Regression:

→ A simple ~~linear~~ linear model that is often used for classification tasks.

→ can be used as a baseline for binary and multi-class classification problems.

ii) Random Forest:

→ An ensemble learning method that combines multiple decision tress.

→ can be used as a baseline for classification and regression problems.

iii) Multinomial Naive Bayes:

→ A simple probabilistic model that is often used for text classification tasks.

→ can be used as a baseline for text classification problems.

iv) k-Nearest Neighbors (kNN):

→ A model that predicts outcomes based on similar data points.

→ can be used as a baseline for classification and regression problems.

2) The Neural N/w Baseline Models are of 3 types:

i) single-layer perception:

→ A simple nn with one i/p layer, one o/p layer and no hidden layers.

→ can be used as a baseline for binary and multi-class classification problems.

ii) Multi-Layer perception (MLP):

→ A simple nn with one i/p layer, one hidden layer & one o/p layer.

→ can be used as a baseline for classification & Regression Prblms.

iii) CNN: → A simple CNN that is often used for image classification tasks.

→ can be used as a baseline for image classificatin prblm

Determining Whether to Gather More data:

→ Determining whether to gather more data in dL involves observing how your model performs & understanding if the existing data is sufficient for it to learn effectively.

Key Measures that need for more data:

i) Underfitting:

If the model performs poorly on both the training and validation sets, it may not be complex enough to learn the underlying patterns, or it may not have enough data to generalize effectively. More data can help the model discover richer representations.

ii) High variance / overfitting:

When the model performs well on the training data but poorly on unseen validation data, it is likely overfitting. While other techniques like regularization and dropout can mitigate overfitting.

iii) Plateaued learning curves:

If the learning curves for both training & validation loss/accuracy have plateaued, and the desired performance hasn't been reached, it might indicate that the model has extracted all the available information from the current dataset. More data could provide new information for futher improvement.

iv) Imbalanced Datasets:
If certain classes or categories are underrepresented in the dataset, the model may struggle to learn them effectively. Gathering more data for these underrepresented classes can improve model performance.

v) Lack of diversity:
If the existing dataset lacks diversity and doesn't adequately represent the real-world scenarios the model ~~makes~~ will encounter, gathering more diverse data is crucial for robust performance.

vi) Specific Error Analysis:
Analyting the types of errors the model makes can reveal specific weaknesses that could be addressed by targeted data collection.

Considerations before gathering more data:

i) cost and effort:
Getting new data & labeling it takes money, time & hardwork. So before collecting more data, we must check the benefits.

ii) Data Quality:
Its not about collecting a lot of data, its about collecting good quality & useful data.

iii) Diminishing Returns:
→ After a point, adding more data will not improve the ~~data~~ model much.
→ The improvement becomes small.

iv) Alternative solns: sometimes you don't need more data instead you can have data agumentation, changing the model design.

Selecting Hyperparameters:

→ Hyperparameters are parameters that are set before training a machine learning model and are not learned from the data. They control how fast the model learns, how complex it becomes and how well it performs.

→ selecting good hyperparameters is important because the wrong set selection can lead to slow training, overfitting, underfitting or poor accuracy.

components for selecting Hyperparameters:

1) Learning Rate:

→ If the learning rate is too high the model will jump too much and never learn properly.

→ If the learning rate is too low, training becomes very slow and may get stuck.

2) Batch size:

→ Batch size means the no. of samples used for each training iteration.

→ small batch size it is slower but more stable learning.

→ Large batch size it is faster but may generalize poorly.

3) No. of Epochs:

→ Epochs tells us how many times the model looks at the whole dataset.

→ If the Epochs are too less, the model underfits.

→ If the Epochs are too high the model overfits.

4) Activation Functions:

choosing activation fn like ReLU, Sigmoid, Tanh, softmax affects how the model learns.

How to select Hyperparametes:

step1: Define the search space:

Identify hyperparameter and set ranges for each hyperparameter, define the possible values or a range to search within.

step2: choose a search strategy:

The search strategy include:

i) Grid search: This method trains the model for every single combination of Hyperparameters you have specified.

ii) Random search: This method selects random combinations of hyperparameters from your defined search space.

iii) Manual search: This involves using your intuition and experience to try different combination

step3: Train and Evaluate models:

Train the model with combinations and Evaluate the performance.

Step 4: Select the best Hyperparameter:

Identify the hyperparameter combination that has best score on validation data and then train your final model using best-performing set of hyperparameters on your entire training dataset.

Debugging strategies:

Debugging dl models requires a systematic approach to identify and resolve issues that hinder model performance or prevent effective learning.

Key strategies:

1) Dataverification:
- Throughly inspect your dataset
- Handle imbalanced datasets
- Verify data preprocessing

2) code and Implementation checks:
- Review your code for errors
- verify model architecture
- Double check loss fn and optimizers
- Inspect weight initialization.

3) Training process Monitoring:
- Track loss curves & Metrics
- Monitor gradients & Activations
- Reduce regularization
- Train with a smaller datasets,

4) Experimentation & Iteration:
- start simple & establish a baseline.
- Hyperparameter tuning.
- cross-validation
- Document & track experiments.

5) Utilizing Debugging Tools & Techniques:

- Use built-in debuggers
- Employ logging & monitoring
- Visualize model o/ps.
- Backtracking & binary search.

Multi-Digit Number Recognition

Recognizing multi-digit numbers is a common deep learning application. The task is to detect multiple digits in an image and classify each digit correctly.

1) Data collection:

- collect images containing multi-digit numbers.
- Each image includes digits in natural scenes with varying illumination, orientation & backgrounds.
- Labels include sequence of digits, not just single numbers.

2) Data preprocessing:-

- Resize images to a fixed size.
- Normalize pixel values (0-1)
- Apply augmentation: rotation, brightness, croping, contrast changes.
- convert labels into digit sequences

Ex: "274" → [2,7,4]

3) Model Design:

Two popular architectures

a) CNN + Multi-class o/p:

- Use a CNN to extract features.
- Use multiple softmax o/ps.

b) CNN + RNN (sequence Model)

- CNN extracts features
- RNN / LSTM / GRU recognizes digit sequences.

4) Training Methodology:

- Split data into train / validation / test
- Use cross-entropy for multi-o/p models.
- Use optimizers & learning rate scheduling.

5) Evaluation Metrics:

- Digit Accuracy - correct digit prediction rate
- Sequence Accuracy - entire ~~no~~ num predicted correctly
- Edit distance - how far prediction is from ground truth
- confusion matrix - Errors b/w similar digits.

6) Deployment:

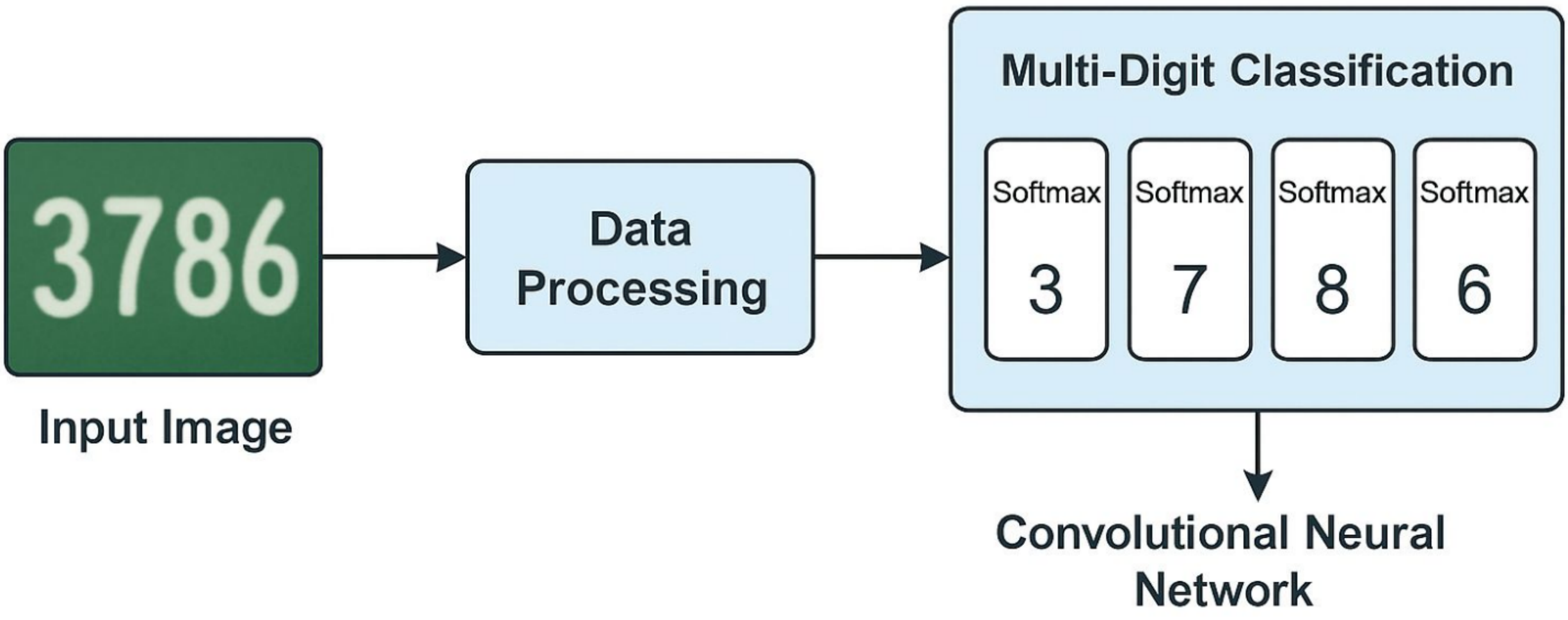
- Export trained model
- Deploy on:
- Mobile apps for meter reading
- edge devices for license plate recognition
- Web servers for automatic form digit extraction
- Real-time inference uses Open CV + DL.

Practical challenges:

i) Digits may overlap or be partially visible.
ii) Background noise & shadows affect accuracy.
iii) Variable number lengths create difficulty.
iv) Real-time apps need fast CNN architectures.

Applications:

- License plate Recognition
- Electricity/water meter reading ~~recognitio~~ automation.
- Bank cheque number extraction
- House number detection
- Handwritten multi-digit OCR.


3786
Input Image
Data Processing
Multi-Digit Classification
Softmax
3
Softmax
7
Softmax
8
Softmax
6
Convolutional Neural Network

Unit - V

1. Large - scale Deep Learning applications :-

i. Google AlphaGO :- Defeated a human world champion in Go, demonstrating the power of deep learning

ii. ImageNet Classification :- Achieved human level performance in image recognition tasks.

iii. Machine translation :- Improved machine translation systems, enabling better communication across languages

2. Computer vision :-

i. self - driving cars :- Companies like Waymo, Tesla, and Uber use computer vision for autonomous driving

ii. Facial Recognition :- used in security systems, social media, and law enforcement.

iii. Medical Diagnosis :- Deep Learning-based computer vision helps in disease diagnosis and medical image analysis.

Speech recognition :-

1) Virtual Assistants :- Amazon, Alexa, Google assistant, and Apple Siri use speech recognition

2) Voice-to-Text :- Servicey like google speech-to and Microsoft Azure speech services.

3) Call centers :- Speech recognition improve customers service and call routing.

NLP :-

1. Chatbots :- used in customer service, tech support, and Conversational AI.

2. Sentiment Analysis :- Companies analyze customer feedback and sentiment on social media.

3) Text - summarization :- Tools like
SummarizationBot use NLP for
Text summarization

Other Applications :-

1. Recommendation :- E-Commerace platforms
system
and streaming service use
deep Learning for recommendation

2. Predictive Maintence :- Industry use deep
Learning for predictive
maintenance and quality
control,

3. Financial Forecasting :- Deep Learning is used
in stock market prediction
and financial analysis