Deep Learning for Computer Vision

NPTEL - DL - Dr. Mitesh Khapra

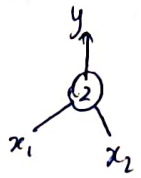
Deep Learning is a subset of AI but fundamentally different from ML

Mc. Culloh Pitts Neuron

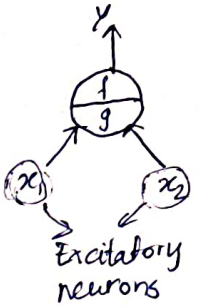
- simplest neuron
- input and output data is binary

1. Aggregation
2. Threshold on aggregation

⇒ Excitatory Neuron



for an AND gate,



Aggregation ⇒ $x_1 + x_2$

⇒ $x_1 + x_2 + \ldots x_n$

⇒ $g(x) = \sum_{i=1}^{n} x_i$

A neuron ~~has both~~ is mathematically $f(g(x))$

Threshold ↑ aggregation ↑

$$f(x) \Rightarrow \begin{cases} 1 & x \geq \theta \\ 0 & x < \theta \end{cases}$$

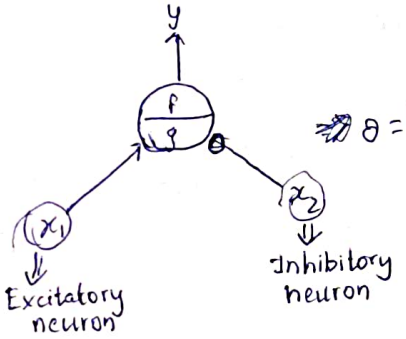
where θ is the threshold.

for an AND, NOT gate ⇒ $x_1 \cdot \bar{x}_2$

x_1	x_2	$x_1 \cdot \bar{x}_2$
0	0	0
0	1	0
1	0	1
1	1	0

Whenever x_2 is 1, the output is 0
⇒ x_2 is inhibitory neuron

⇒



⇒ $\theta = 1$ for AND with NOT gate

Inhibitory Neuron

⇒ if the inhibitory neuron returns an output of 1, the result is always 0.

- it doesn't participate in aggregation

The type of neuron is identified by its associated weight.

for excitatory neuron, weight is 1

for inhibitory neuron weight is -1

Perceptron

$$g(x) = w_1 x_1 + w_2 x_2 + w_3 x_3 + \ldots + w_n x_n = \sum_{i=0}^{n} w_i x_i$$

$$= WX$$

$$\sum_{i=1}^{n} w_i x_i - \theta = g(x) \Rightarrow f(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

$$g(x) = \sum_{i=0}^{n} w_i x_i \quad \text{where } w_0 = -\theta \text{ and } x_0 = 1$$

For an XNOR gate,

x_1	x_2	$\overline{x_1 \oplus x_2}$
0	0	1
0	1	0
1	0	0
1	1	1

and weights.

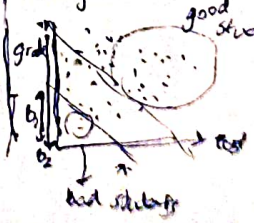
in a perceptron, inputs can be any real value.

$W = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix}$ $X = [x_0 \; x_1 \ldots x_n]$ → 1 x n matrix

→ n x 1 matrix

understanding of bias

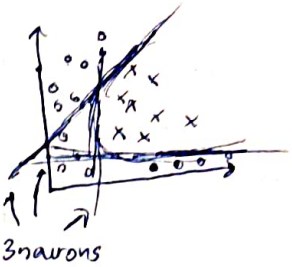
Suppose a college is taking students for admission based of grades and test results, new colleges may be alright with lower values than the good ones

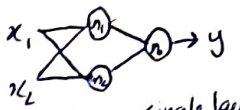


Linear separability

- A single neuron can only separate 2 classes with a straight line
- in order to separate 2 classes with a curve, multiple lines are used by combining multiple neurons and forming a neural network

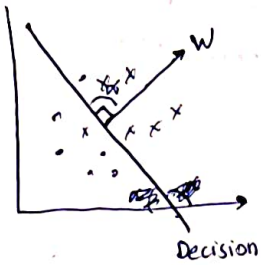
$w_0 = -1$
$w_1 \geq 1$
$w_2 < 1$
$w_1 + w_2 \geq 1$





simple single layered neural network.

Learning Logic



$W \cdot X = 0 \Rightarrow$ decision boundry. (W is orthogonal to decision boundary)

The decision boundary is the line (imaginary) which classifies points into class 1 or class 2.

if P is positive but classified as -ve then $W = W + X$ (repositioning the D.B. to make the correct classification)

$\cos\alpha = \frac{X \cdot W}{\|W\| \|X\|} \Rightarrow \alpha_{new} < \alpha$

↳ The new angle b/w W and D.B. is less than the previous angle.

angle b/w W and D.B. on updating W.

$\alpha_{new} > \alpha$ for when P is -ve but classified as negative.

Perceptron learning algorithm

Aggregate = $\sum_{i=1}^{n} w_i x_i + \text{bias} = y_{in}$

Activation function => bipolar step activation =>

$$\begin{cases} 1 & y_{in} \geq \theta \\ 0 & -\theta \leq y_{in} \leq \theta \\ -1 & y_{in} < -\theta \end{cases}$$

compare if $y \neq t$

$w_{new} = w_{old} + \alpha \cdot x \cdot t$

$bias_{new} = bias_{old} + \alpha \cdot t$

$\alpha \Rightarrow$ learning rate
$w \Rightarrow$ randomly initialised weights
$t \Rightarrow$ target result / expected result.

even though the error is not considered, and simply the expected output is used to adjust the weights, the perceptron will converge as the inputs are meant to be simple

t can only be +1 or -1. (bipolar-class classification)

2) Implement AND with bipolar inputs and outputs, assume $\alpha = 1$, $W_{in} = [0, 0, 0]$

x_1	x_2	$\alpha=1$	t	y_{in}	y	Δw_1	Δw_2	Δb	w_1	w_2	b
									0	0	0
1	1	1	1	0	0	1	1	1			
1	-1	1	-1	1	1	-1	1	-1	1	1	1
-1	-1	1	-1	2	1	1	-1	-1	0	2	0
-1	-1	1	-1	-3	-1	0	0	0	1	1	-1
1	1	1	1	1	1	0	0	0	1	1	-1
1	-1	1	-1	-1	-1	0	0	0	1	1	-1

$\theta = 0$
$\alpha = 1$

Q> AND NOT function ; bipolar inputs and outputs

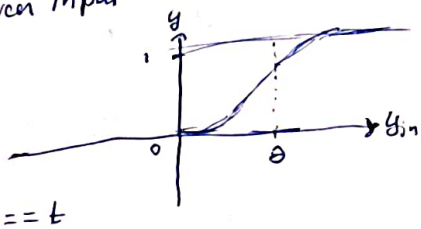
$x_1 \cdot \bar{x_2}$

$w_{new} = w_b + \alpha \cdot x \cdot t$
$\Delta w = \alpha x \cdot t$
$\Delta b = \alpha t$

x_1	x_2	α	t	y_{in}	y	Δw_1	Δw_2	Δb	w_1	w_2	b
1	1	1	-1	0	0	-1	-1	-1	0	0	0
1	-1	1	1	-1	-1	1	-1	1	-1	-1	-1
-1	1	1	-1	-2	-1	0	0	0	0	-2	0
-1	-1	1	-1	2	1	1	1	-1	0	-2	0
1	1	1	-1	-1	-1	0	0	0	1	-1	-1
1	-1	1	1	1	1	0	0	0	1	-1	-1
-1	1	1	-1	-3	-1	0	0	0	1	-1	-1
-1	-1	1	-1	-1	-1	0	0	0	1	-1	-1

Drawback of step functions => - There is no explainability on why the output is 0 or 1 or -1
- ~~it is difficult to understand~~
- There is a hard-threshold at a given input

~~Draw~~



Hence, we begin using sigmoid input function.

$sig(x) = 0 \text{ to } 1 = \frac{1}{1+e^{-y}} = \frac{1}{1+e^{-(w_1+b)}}$

Loss function is calculated instead of only checking for $y == t$

Another reason for not using step function is the difficulty in convergence.

Vanishing gradient => $\Delta y \approx 0$ for $\Delta y_{in} \neq 0$

Sigmoid function gives probability of a test case being classified as positive

i.e.- suppose the sigmoid output is 0.7 for a test case => There is a 70% chance of the point being classified as belonging to positive class

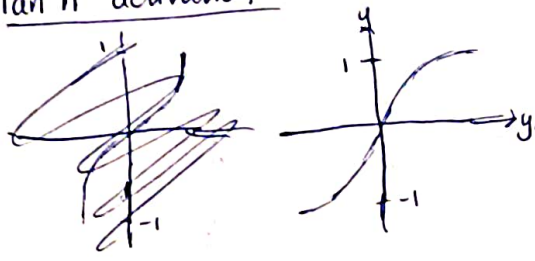
Probability of being in negative class is taken as $1-P$

Softmax function is an alternative to sigmoid which is used for multi-class classification

$\hookrightarrow \frac{e^{z_i}}{e^{z_1} + e^{z_2} + .. e^{z_n}}$ where z_i => probability for i^{th} class.

(sigmoid function is not to be used for multi-class classification)

Tan h activation



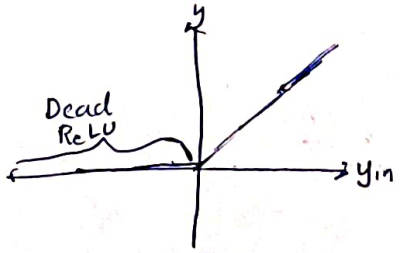
- This activation function is used when even the neutral (0) is considered in classification
- Doesn't give probability
- computationally intensive $\frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Vanishing Gradients is still an issue

Rectified Linear Unit (ReLU)

$ReLU(x) = \max(0, x)$

- no vanishing gradient
- computationally simple

ReLU is used when lot of layers are involved due to simplicity in computation.



SiLU ⇒ sigmoidal + ReLU
- simple computation
- used in recent YOLOs

Loss Function

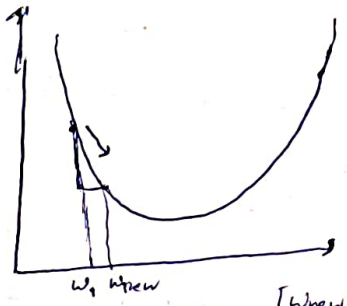
- Binary cross-entropy loss / o/1 log loss = $\sum y_i \log P_i + (1-y_i)\log(1-P_i)$ ← p_i - class classification
  - loss is either 0 or 1
  - $\sum y_i \log P_i$ → multiclass classification
- Mean Absolute Error
- Mean Squared Error
- Root Mean Squared Error
- Categorical cross entropy

Gradient Descent

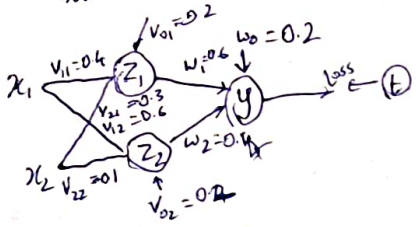
$$\Delta W = \frac{\Delta L}{\Delta W}$$

$$W_{new} = W_{old} - \alpha \Delta W$$

$$b_{new} = b_{old} - \alpha \Delta b$$



ΔW_1 < 0 [$w_{new} - w_i$]
↳ to reach w_{new} from w_1, we need to do subtraction



$$M.S.E. = \frac{1}{2}(t-y)^2$$

Sigmoid ⇒ $\sigma(x) = \frac{1}{1+e^{-x}}$

$x_1 = 1$

$x_2 = 1$

$$Z_{1in} = x_1 v_{11} + x_2 v_{21} + v_{01} \times 1$$
$$= 1 \times 0.4 + 1 \times 0.3 + 0.2$$
$$= 0.9 \Rightarrow Z_1 = \sigma(0.9) = 0.5744$$

$$Z_{2in} = x_2 v_{22} + x_1 v_{12} + v_{02} \times 1$$
$$= 1 \times 0.1 + 1 \times 0.6 + 0.2 \times 1$$
$$= 0.7 \Rightarrow Z_2 = \sigma(0.7) = 0.6684$$

$$y_{in} = 0.5744 \times 0.6 + 0.6684 \times 0.4 + 1 \times 0.2 = 0.8788$$
$$y = \sigma(0.8788) = 0.7065$$

Compute values to 4 decimal places

suppose $t = 1$

$$Loss = \frac{1}{2}(t-y)^2$$

$$\Delta W = \frac{\delta L}{\delta w} \Rightarrow \frac{\delta\left(\frac{1}{2}(t-y)^2\right)}{\delta w}$$

there is no 'w' term here but in y, there is a y_{in} and in y_{in} there is w term

$$\Rightarrow \frac{\delta L}{\delta w} = \frac{\delta L}{\delta y} \times \frac{\delta y}{\delta y_{in}} \times \frac{\delta y_{in}}{\delta w}$$

$$\frac{\delta\left(\frac{1}{2}(t-y)^2\right)}{\delta y} = 2 \times \frac{1}{2} \times -(t-y)(1) = y - t$$

$$\frac{\delta y}{\delta y_{in}} \Rightarrow \frac{\delta\left(\frac{1}{1+e^{-y_{in}}}\right)}{\delta y_{in}} = \frac{\delta(1+e^{y_{in}})^{-1}}{\delta(1+e^{y_{in}})} \cdot \frac{\delta(1+e^{y_{in}})}{\delta y_{in}}$$

$$= \frac{e^{-y_{in}}}{(1+e^{-y_{in}})^2} \Rightarrow \underbrace{\frac{1}{1+e^{-y_{in}}}}_{y} \times \underbrace{\frac{e^{-y_{in}}}{1+e^{-y_{in}}}}_{(1-y)}$$

$$\frac{\delta y_{in}}{\delta w} = \frac{\delta(w_1 z_1 + w_2 z_2 + w_0 z_0)}{\delta w_i}$$

$= z_i$ ← based on which weight we are updating, rest are considered as constants.

$$\Rightarrow \frac{\delta L}{\delta y} \times \frac{\delta y}{\delta y_{in}} \times \frac{\delta y_{in}}{\delta w} = (y-t)(y)(1-y)(z_{in})$$

$$\Rightarrow$$

in example, $\Rightarrow (0.7065 - 1)(0.7065)(1 - 0.7065)(0.5744)$

$\Rightarrow \Delta w_1 = -0.0349$

$w_1 = 0.6 - 0.25(-0.0349)$

$= 0.6087$

→ precompute this value

$\Delta w_2 = (0.7065 - 1)(0.7065)(1 - 0.7065)(0.6684)$

$= -0.04067$

$w_2 = 0.5 - (0.25)(-0.04067)$

$= 0.5102$

Don't use updated values for updation anywhere until all values are updated.

$\Delta w_0 = -0.0608$

$w_0 = 0.2 - 0.25(-0.0608)$

updating v_{11}, v_{12}, v_{21}, v_{22}

$$\frac{\delta L}{\delta v_{11}} = \frac{\delta L}{\delta y} \times \frac{\delta y_{in}}{\delta y_{in}} \times \frac{\delta y_{in}}{\delta z_1} \times \frac{\delta z_1}{\delta z_{1in}} \times \frac{\delta z_{1in}}{\delta v_{11}}$$

(similar differentiation to $\frac{\delta y}{\delta y_{in}}$)

$\frac{\delta y_{in}}{\delta z_1} = w_1$

$\frac{\delta z_1}{\delta z_{1in}} = (z_1 \times 1 - z_1)$

$\frac{\delta z_{1in}}{\delta v_{11}} = x_1$

$= (y-t)(1-y)(y)(w_1)(z_1)(1-z_1)(x_1)$

$\Rightarrow \Delta v_{11} = -0.0023$

$v_{11} = 0.4 - 0.25(-0.0023) =$

$\Delta v_{21} = (y-t)(1-y)(y)(w_1)(z_1)(1-z_1)(x_2) = -0.0023$

03/02/2025

- There is no difference b/w ANN and DNN except that a DNN has many more hidden layers.
- All YOLO models are trained on COCO datasets.
- Take a pretrained model for any purpose, as they are guaranteed to have good feature extractors. (it is assumed that they have the best hyperparameters.)

Then, remove its output layer since we want a different classifications than what it was originally trained for. Then we replace it with our own neural network.

Fine tuning a model for a purpose => appending your own neural network to the output of the pretrained model. (Transfer Learning and fine Tuning)

Add a good number of layers, not just 1 or 2.

When appending your own neural network,

- never keep sudden decrease in number of neurons.

i.e.- Pretrained model → 1024 neurons → 2048 neurons → 1024 neurons → 2 neurons

sudden convergence will cause difficulty in learning.

Training a model

Dataset → Training (60% ~ 70%)
→ Validation (15% ~ 20%)
→ Test (15% ~ 20%)

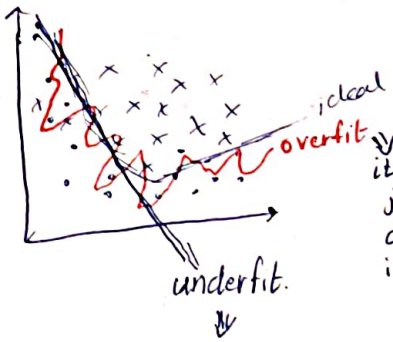
Bias ⇒ difference b/w mean of expected and calculated mean
= $E(f(x)) - f(x)$

Theoretically calculated mean. i.e.-

Variance ⇒ how much the calculated values/lines vary from the expected line.
$= E([f(x) - E(f(x))]^2)$

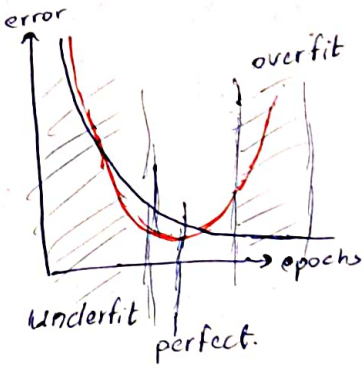
Complexity of model depends on multiple factors
- no. of layers
- no. of neurons
- no. of epochs of training

too much complexity is as bad as too less complexity



it is like joining the data points and imitating the data

For each epoch,
1. Training on 1 epoch
2. validation on validation data
- continue till training loss decreases but validation is increasing → overfitting.



Early stopping

Avoiding overfitting has many different methods.

A fundamental technique used to avoid overfitting.

Dropouts ⇒ Randomly dropping out neurons based on some probability. → in internal layers.

It is to avoid problem of very few of the neurons learning the correct structure while rest remain useless. This method causes neurons to forcefully learn.

Having 2 consecutive dropouts before a layer simply means that the probability of dropout is increased for the next layer.

Handling Overfitting [based on number of epochs]

- early stopping
- Dropouts
- Data augmentation
  - When data is small and doesn't contain enough variations, the model is likely to overfit. So, in order to increase variations, properties of the image like contrast, brightness, scale, etc. are changed. mosicing
  - The chosen data augmentation needs to be appropriate to the dataset.
- Ensemble learning
  - Using different training sets of data for multiple models that are identical
  - ideally, assuming that the error in prediction by each of the models is independent of each other, error = $\frac{1}{N} \times$ error of each model
- Adding noise
- Regularization
- Batch Normalization

Geometric data augmentation causes issue in YOLO models as the coordinates of bounding boxes are definitely going to change.

Regularization

- L_1 regularization
- L_2 regularization

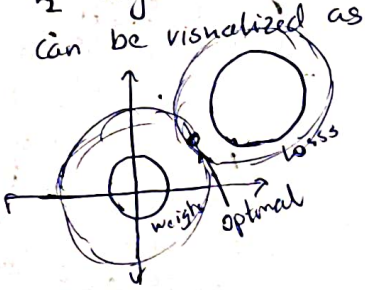
Instead of minimizing error, we minimize True error (target - pred)/N

L_2: T.E. = Error + $\alpha \|w_i\|$ (a.k.a. Empirical error)

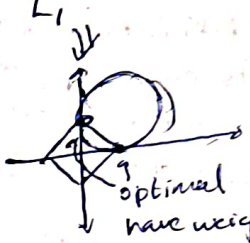
- In order to make sure that the weights don't shoot up.
- When weights shoot up, it can be said that the model is not able to find clear patterns within the data due to which the weights constantly change.
- Penalizing the weights => keeping the weights in check.
- L_1 is used for dimensionality reduction and not for ~~er~~ overfitting handling. L_2 is for overfitting.

True error can also be written as T.E. = Error + $\alpha\|w_i\|$

the equation of L_2 regularization can be visualized as



L_1



Adding Noise

- we assume that the noise follows a Gaussian distribution
- ε_i represents the noise (random variable)
- $\tilde{x}_i = x_i + \varepsilon_i$ => ~~data~~ inputs with added noise.
- $\tilde{y}_i = \sum w_i \tilde{x}_i$ => $E(\tilde{y}_i - y_i) = E((\hat{y}_i - y) - \sum w_i \varepsilon_i)^2$ (expected)

$= E(\hat{y} - y)^2 + (\sum w_i \varepsilon_i)^2 + 2E((\hat{y}-y)\sum w_i \varepsilon_i)$

$\Downarrow$

same as L_2 regularization $\Leftrightarrow$ Error + $\alpha\|w_i\|^2$

γ → because covariance between ε_i and $\hat{y} - y$ is ~~zero~~ 0.

Training a YOLO model:-

1. install YOLOv... using pip
2. Have a dataset that ~~is valid~~ for is compatible with YOLO
3. Use command line of YOLO to train model.

Batch normalization

=> Creating batches of data and applying normalisation of each.

Suppose we take a batch of data, for each attribute within a row, we calculate the mean and ~~standal~~ standard deviation.

=> $\tilde{A}_i = \frac{A_i + \mu}{\sigma}$ —(scale / shift)→ $A_i \gamma - \alpha_i + \beta$ → within each batch

$\Downarrow$ moving avg. across batches

$\mu_{mov} = \gamma \mu_{mov} + (1-\gamma)\mu_i$

$\sigma_{mov} = \gamma \sigma_{mov} + (1-\gamma)\sigma_i$

Convolutional Neural networks are designed to work with grid-like structures.

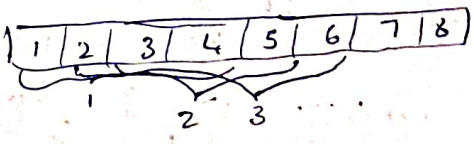
1-D CNNs can be used to work on time-series data.

What is convolution?

Suppose we have a 1-D array of 8 elements, and we have a function that accepts 4 inputs and gives 1 output. (kernel)

Convolution is taking 4 inputs at a time

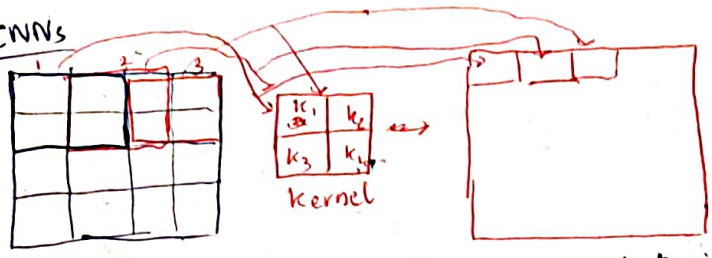
- uses concept of sliding window



in a convolutional neural network,
convolution of input and filter is used to compute output.
↳ * ↳ x ↳ w → also known as kernel (synonymous with the weights in normal neural networks)

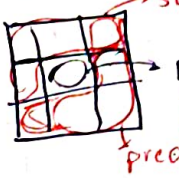
$$s_t = (x * w)_t$$

2D CNNs



CNN is not much different from ~~CNN~~ ANN, basically the inputs are aggregated by convoluting it with the kernel and summing it all up to predict a value for that given cell in the matrix.

the input is generally taken as 3x3 or nxn where n is odd



A value is set for this pixel based on surrounding cell values; ~~as well as~~ ~~the current~~ and the ~~current~~ kernel.

Basic example of convolution ⇒ image editing (blurring, brightness, etc.)

$$\begin{bmatrix} \cdots \\ \cdots \\ \cdots \end{bmatrix}_{n \times n} * \begin{bmatrix} k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 \\ k_7 & k_8 & k_9 \end{bmatrix}_{F \times F} = \begin{bmatrix} \cdots \\ \cdots \\ \cdots \end{bmatrix}_{n \times n}$$

(when there is padding at ends)

2D matrix of multiple values

Kernel FxF

output 2D convoluted matrix

Types of convolution

1. No-zero padding / valid convolution ⇒ not padding with zeroes on edges of matrix to help fit the kernel function such that the cells on edges are aligned with
2. Just enough zero Padding / Same convolution

3. Full convolution ⇒

4. Stride (S) ⇒ when using the sliding window, instead of shifting sliding window by 1, we shift it by the value of S, a.k.a. stride. Doing this reduces the dimension of the output as well.

$$W_2 = \frac{W_1 - F + 2P}{S} + 1 \qquad H_2 = \frac{H_1 + - F + 2P}{S} + 1$$

↳ Done with the assumption that images have high definition. Since the image has high definition the next assumption is that the neighbouring pixels share same/similar information so we can ignore them. Resulting image can be called as a downsampled image. The process of reducing dimensions of the image is called down sampling.

5. Depth (K) - For an image, we can have multiple dimensions (R, G, B). There will be 1 2D matrix kernel for each dimension of the image. The number of such kernels is basically the depth of the kernel

⇒ input ⇒ 227 x 227 x 3 kernel ⇒ 11 x 11 x 3 Filters = 96 Stride = 4 Depth = 96

$\frac{216}{4} = 54$ $\frac{227 - 11}{4} + 1 = 55$ ⇒ 55 x 55 x 96 output.

An ANN has Dense connections (every neuron is connected to every other neuron)
But a CNN has sparse connections (not every neuron is connected to every other neuron)

ANNs will also work on images however, the dense connections of the neurons because the ANN to expect similar input every time. However, the image that is passed into the ANN may be slightly shifted / scaled / rotated which confuses the ANN.

Therefore ANN will work on images of specified formats only, which make them unsuitable to work with images.

CNN does <u>parameter sharing</u>

Towards the end of a convolutional network, the connections are <u>dense</u>.

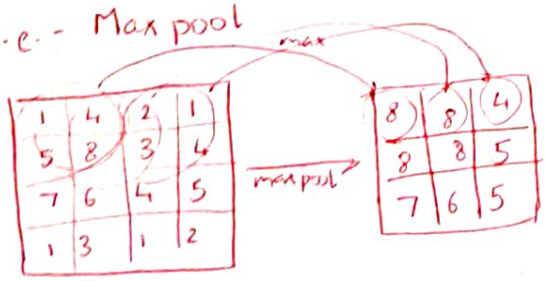
input → convolution → Activation → Max pool layer → Flatten layer → Fully connected Layer ReLU activation function → Softmax o/p

Stride 2 → used to down-sample

Pooling

- helps to fetch statistics of data in the neighbourhood
  - ↳ Helps to reduce computations while minimizing loss of data.

i.e. - Max pool



Max pooling helps to make the model scale-invariant.

Usually, pooling is used along with striding so that there is down sampling

LeNet (an example CNN)

- ↳ Developed to extract postal codes
- ↳ Input => 32 x 32, black and white only.

k => depth of output, not kernel.
no. of parameters => dimensions of kernel x k + bias

Input: A, 32 x 32

Convolution layer 1: S=1, F=5, P=0, K=6 → params = 140

Pooling layer (14x14): S=2, F=2, K=6, P=0, params=0 (because no parameters are being fetched here)

Convolution layer 2 (10x10): S=1, F=5, K=16, P=0, params = 2400 (5 x 5 x 6 x 16)

pooling layer 2 (5x5): S=2, F=2, K=16, P=0, params = 0

Flatten → Fully connected layer (120 neurons), params = 48120 → Fully connected layer 84 neurons → Fully connected 10 neurons → output

ILSVRC (ImageNet Large Scale Visual Recognition Challenge) → 1,281,167 images (ImageNet)

↳ When a model is proposed for object classification, it needs to be trained on this dataset and prove its accuracy in classification

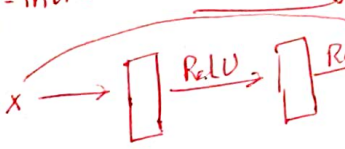
VGGNet

⇒ aims to minimize no. of parameters
⇒ Same kernel size throughout the network — principle ⇒ Applying 2 3×3 kernels consecutively produces same size output as a 5×5 kernels

↳ By reducing parameters, the computation within each layer can be reduced.

Residual Networks (ResNet)

- introduces Identity connections



$H(x) = f(x) + x$ (output)

[we are taking residue from previous layer]

⇒ Helps to overcome vanishing gradient issue.

↳ The input is passed along with output of previous layers into the upper layers of the network.

State space models

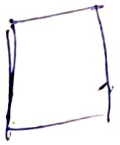
Region Mamba

RCNN ⇒ Region based CNN ⇒ Detection + classification
YOLO ⇒

↳ 2 stages
↳ 1. selecting bounding boxes
↳ 2. Classification of object in bounding box. (using CNN)

YOLO is a single stage detector; i.e., it takes the object bounding box and search for class of object, only during training stages.

Neck, backbone of model.



↑
Feature extraction

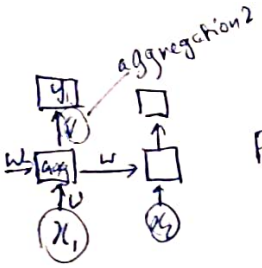
Recurrent Neural Network

Many to many ⇒ language 1 to language 2 translation
Many to one ⇒ sentiment analysis
one to many ⇒ generation of data.

$$s^{(t)} = f(s^{(t-1)}, \theta)$$

$s^{(t)}$: current state; $s^{(t-1)}$: previous state; θ: current input

In RNN, the output is not passed in as input again, the state is passed again to the next cycle of input



Parameters are shared
↳ Same parameters (weights) are used for processing all different inputs.

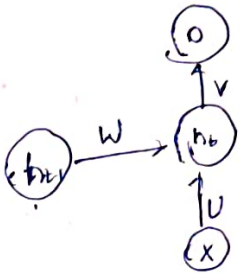
Aggregation (of inputs and history) → Activation → Aggregation again → Activation → output (softmax)

applied to each element within the aggregated value

$o = g(c + Vh_t)$

$h_t = f(Wh_{t-1} + Ux + b)$

Training the network



backpropogation

$V = V - \frac{\partial L}{\partial V}$

$$\frac{\partial L}{\partial V} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial o} \cdot \frac{\partial o}{\partial V}$$

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial o} \cdot \frac{\partial o}{\partial h_t} \cdot \frac{\partial h_t}{\partial W}$$ ⇒ requires $h_{t-1}, h_{t-2} \dots h_0$

↳ back propogation through time

$h_t = \tanh(b + Wh_{t-1} + Ux_t)$

~~iterating using history and weights~~

to update the weight W, we need to consider all previous history.

$$\frac{\partial L}{\partial W} \sum_{r=1}^{t} \frac{\partial h_t}{\partial h_r} \cdot \frac{\partial h_r}{\partial W}$$

make a vector/stack of

Types of RNN applications

1. One to one
2. One to many ⇒ one ~~example~~ input to give a sequence of outputs.
3. Many to one ⇒ a sequence of inputs is given and a single output is given.
4. Many to many ⇒ sequence of inputs → sequence of outputs.

Disadvantages

1. Vanishing Gradients (in case the weights are too small ~~&~~/negligible)
2. Exploding Gradient (in case the weights are too big)

LSTM ⇒ extension of RNN where the memory that is important in long term and short term is used in current pred.

Suppose x_t ⇒ input,
$\hat{y}_t$ ⇒ predicted output

$s_t = W \cdot a_{t-1} + U x_t + b$ (W is a matrix)

$a_t = \tanh(s_t)$

$o_t = V \cdot a_t + c$ (another matrix)

$\hat{y}_t = \text{softmax}(o_t)$ ← prediction

example input and output

	in	out
1	0	0.5
2	0.5	1
3	1	0.5
4	0.5	0

take $a_0 = [0.5]$ (1×1 matrix)

$W = [1]$ $b = -1$

$x_t = [0]$

$U = [0.3]$

$s_t = [0.5] + [0] - [1]$

$= [-0.5]$

$a_t = [\tanh(-0.5)] = [-0.008]$

$V_1 = 0.3$

$o_t = [0.3 \cdot (-0.008)] + [0.1] = [0.0976]$

$\hat{y} = \sigma(0.0976)$

$= 0.5243$

loss $= 0.5 - 0.5243$

$= -0.0243$

LSTM

based on input, the memory of the previous state is retained.

input, output, forget gates

taking example of NLP, (sentiment analysis)

we would
1. Forget stopwords
2. Selectively read information having sentiment words
3. selectively write

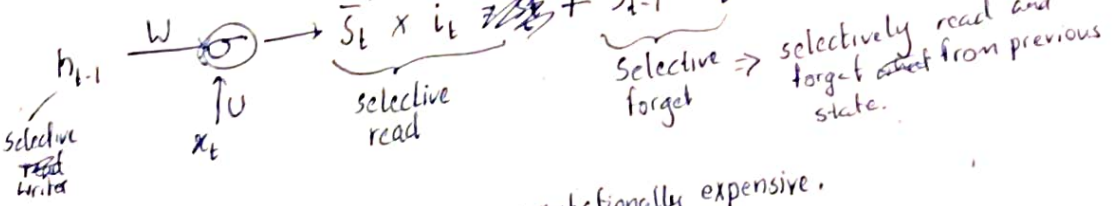
S_{t-1} => Book + 60 slides $\times$ O_t $= h_{t-1}$ => 40 slides → process is called selective write

O_t ↓ decides what to remember (a matrix) ↓ called output gate

$O_{t-1} = \sigma(W_o h_{t-2} + U_o x_{t-1} + b_o)$

Suppose we want to caption a video, we make changes in RNN.

RNN (CNN)



This model of an LSTM is very computationally expensive.

Hence introduction of Gated Recurrent Units (GRU)

↳ in # gated recurrent units, the forget gate is basically taken as $1 - i_t$ input gate.

↳ GRU is optimized on computation hence may perform worse than LSTM but not better than LSTM.

GRU may or may not train faster than LSTM

Bi-LSTM => bidirectional LSTM

Encoders and Decoders

Encoders and decoders can be anything, like RNN, LSTM, etc.

For example, suppose an image is to be converted to text,
- we first have a CNN to be encoder => Extract features of the image
- then we have another model, i.e. RNN to generate text from the input feature.

Applications of Encoder-decoder
- Transliteration
- Text summarization

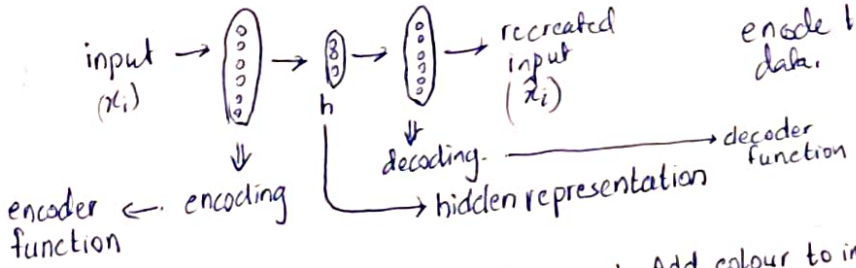
Textual entailment => based on text 1, text 2 is constructed.

2 RNNs are used, output of 1 RNN is used as input to next RNN

Encoder, decoder is different from U-net and Transformer

Autoencoders

↳ Used for PCA
↳ A neural network that takes x as input, encodes, decodes and gives out x as output



encode to create a latent representation of the data.
↳ compressed.

Autoencoders can be used to → 1. Add colour to images
↳ 2. Add clarity to images. (~~add~~ noisy image as input, loss is to be calculated as loss of data b/w original and predicting)

Undercomplete Autoencoder - hidden layer size is smaller than input layer size.

Overcomplete Autoencoder - hidden layer size is larger than input layer size
↳ should have non-linear function in order to ensure that the neural network is learning instead of acting as identity functions.

Reconstruction error ⇒ error b/w input and recreated input (output)
↳ Used when input data is very complex and some extra data would help to identify dependancies in the data.

Regularized Autoencoder (L2 regularization)

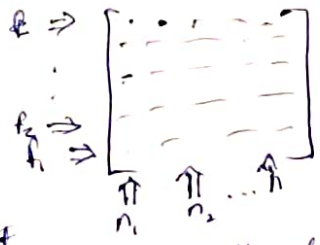
↳ Train any architecture of autoencoder
↳ Freely change the dimension of the hidden representation

Sparse Autoencoder (for overcomplete autoencoder)

↳ similar to dropouts is done.
↳ ~~Hidden~~ Neurons are dropped in hidden layer so that ~~only~~ same number of neurons are present in HL and input layer.

Contractive Autoencoders

↳ Also uses regularization
↳ A matrix is created ~~such~~ storing the change in output of each neuron with respect to each feature
↳ computing jacobian coefficient for each feature
↳ used to remove unimportant features.



Denoising Autoencoders

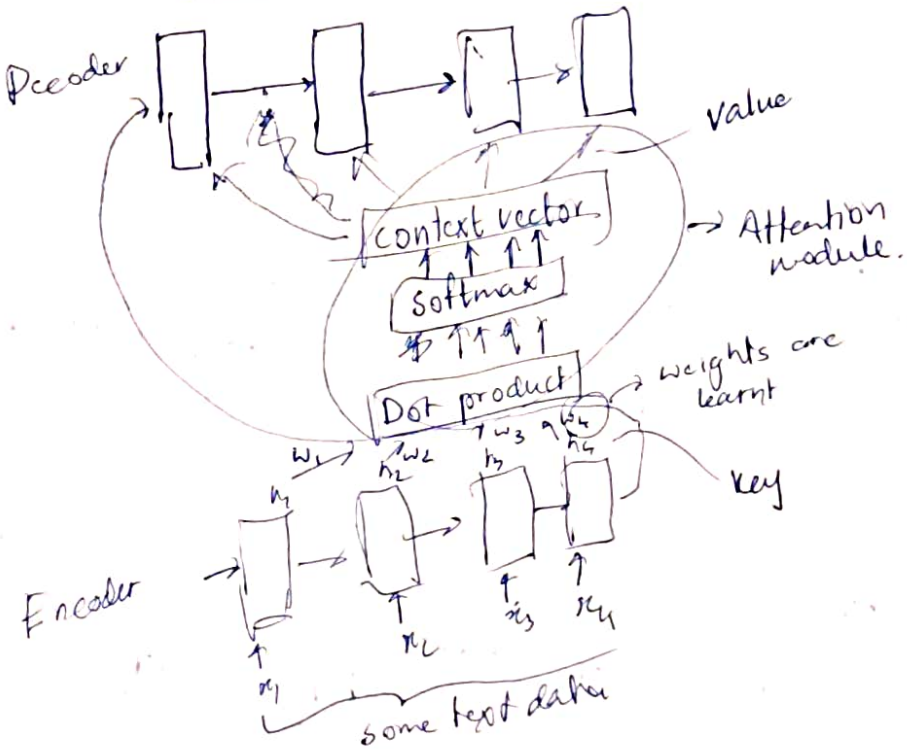
↳ Adding Gaussian noise to the input data while having ~~predicted~~ output to be image without noise. Hence using the ~~data~~ model to learn to remove the noise.

DL

Attention

Find relevance of current content (query) w.r.t. previous states. → key

Attention and Transformers were originally made for NLP

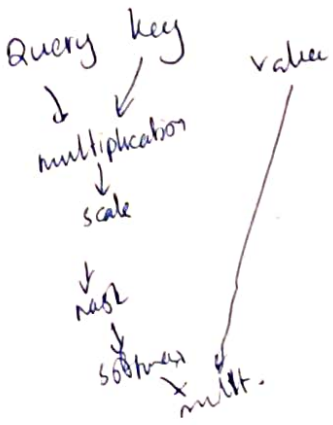


$$\frac{(Q \cdot K)V}{\sqrt{a_K}}$$

Self attention

↳ find relevance of part of current context w.r.t. another part in the same sentence

Transformers are basically use of multiple self-attention module



The model may have to train for more epochs after attention is added