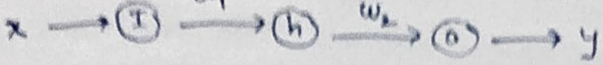
* Back ... resent problem and updating the wts.



$w_1 = 0.15$ $x = 5$

$w_2 = 0.4$ $y = 1$

$\eta = 0.5$

learning rate

solution:-

Forward Pass → input data is fed thro the network, layer by layer to produce an output prediction.

(a) h

$net_h = w_1 x = 0.15 \times 5 = 0.75$

$out_h = \frac{1}{1+e^{-net_h}} = \frac{1}{1+e^{-0.75}} = 0.679$

error fn is calculated based on pred^n & actual target

(b) O

$net_o = w_2 out_h = 0.4 \times 0.679 = 0.272$

$out_o = \frac{1}{1+e^{-net_o}} = 0.568$

measures how far ntw is from making correct predictions

$E = \frac{1}{2}[y - \hat{y}]^2 = \frac{1}{2}[1 - 0.568]^2 = 0.093$ error as $y = 1$

ntk error

- Backpropogation:- → error is propogated backward thro the nwk. Starting from output layer & moving to input layer.

(a) $w_{2\,new} = w_{2\,(old)} - \eta \frac{dE}{\partial w_2}$

During this, partial deri. of error wrt wt & bias is calculated using chain rule. (eff. computation of gradient for all parameters in network)

$\frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial out_o} \times \frac{\partial out_o}{\partial net_o} \times \frac{\partial net_o}{\partial w_2}$

$\frac{\partial E}{\partial out_o} = \frac{\partial \left[\frac{1}{2}(y - \hat{y})^2\right]}{\partial out_o} = \frac{y - out_o}{out_o} \times -1 = 1 - 0.568 \times -1$

$= -0.438$

$\frac{\partial out_o}{\partial net_o} = \frac{\partial \left(\frac{1}{1+e^{-net_o}}\right)}{\partial net_o}$ 1° derivative of S fn ⇒ A·F(1−A·F)

$= out_o(1 - out_o) = 0.568(1 - 0.568)$

$= 0.245$

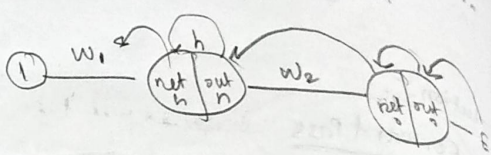
$$\frac{\partial net_o}{\partial w_2} = \frac{\partial w_2 out_h}{\partial w_2} = out_h = 0.679$$

$$\frac{\partial E}{\partial w_2} = -0.432 \times 0.245 \times 0.679 = -0.072$$

Once gradients are calculated, an optimization algo uses them to update the network's wts & biases iteratively reducing the error & improving the network's performance.

$$w_{2\,new} = 0.4 - \eta \frac{\partial E}{\partial w_2} = 0.4 - 0.5(-0.072) = 0.436$$

(b) $$w_{1(new)} = w_{1\,old} - \eta \frac{\partial E}{\partial w_1}$$



$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial out_o} \times \frac{\partial out_o}{\partial net_o} \times \frac{\partial net_o}{\partial out_h} \times \frac{\partial out_h}{\partial net_h} \times \frac{\partial net_h}{\partial w_1}$$

already done

$$\frac{\partial net_o}{\partial out_h} = \frac{\partial (w_2 out_h)}{\partial out_h} = w_2 = 0.4$$

$$\frac{\partial out_h}{\partial net_h} = \frac{\partial \left(\frac{1}{1+e^{-net_h}}\right)}{net_h} = out_h(1-out_h)$$

$$= 0.679(1-0.679) = 0.218$$

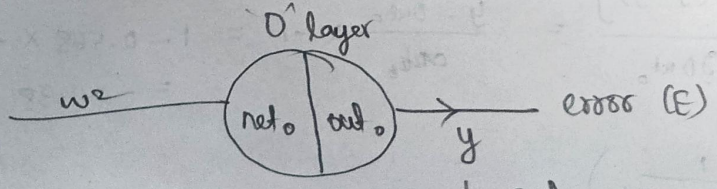
$$\frac{\partial net_h}{\partial w_1} = \frac{\partial w_1 \times x}{\partial w_1} = x = 5$$

$$\frac{\partial E}{\partial w_1} = -0.438 \times 0.245 \times 0.4 \times 0.218 \times 5 = -0.046$$

$$w_{1(new)} = w_{1(old)} - \eta \frac{\partial E}{\partial w_1} = 0.15 - 0.5 \times -0.046$$

$$= 0.15 + 0.023 = 0.173$$

$w_{1(new)} = 0.173$
$w_{2(new)} = 0.436$
} new values



UNIT-3

* Error function :- (Loss function)

~~- disperancy~~

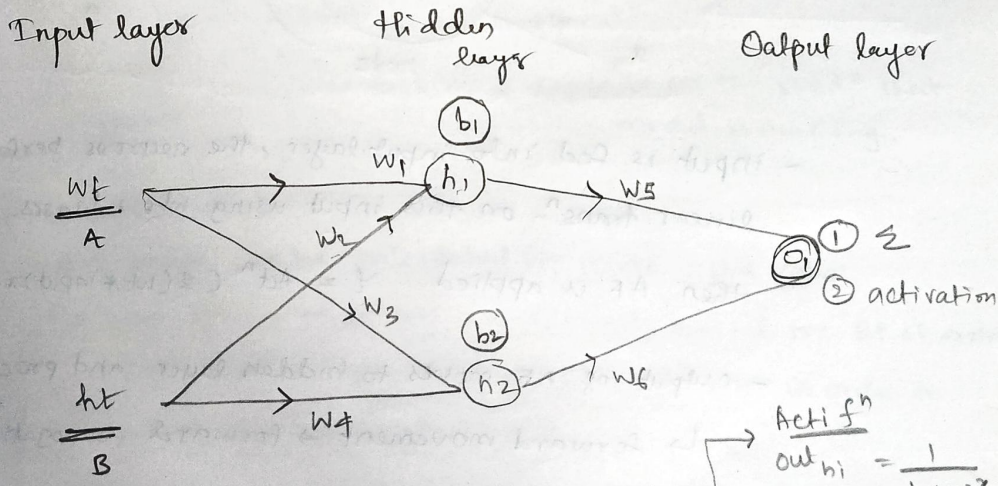
- discrepancy b/w the predicated output of NN & actual target output for a given input
- provide a measure of how well the network is performing.

Common error f^n's include :-

→ MSE : used in regression problems
- calculate avg of sq. differences b/w predicted & actual values.

→ Cross - entropy ~~values~~ loss :
- classification problems (Multi-class)
- diff b/w predicted prob. distribution & true distribution

→ backward propogation of errors :
- used to eff. compute the gradients of ~~error~~ loss w.r.t wts & biases of NN
- these gradients indicate the dir & mag by which the wts & biases shld be adjusted to minimize the error.



$net\ h_1 = W_1 A + B W_2 + \delta_1$

$net\ h_2 = W_3 A + B W_4 + \delta_2$

Activ f^n

$out_{hi} = \frac{1}{1+e^{-x}} = \frac{1}{1+e^{-net_{h_1}}}$

$\frac{1}{1+e^{-net\ h_2}}$

$net\ o_1 = w_5\ out\ h_1 + w_6\ out\ h_2 + \delta_3$

* Activation f^n s (AF) - linear, Softmax, Tanh, ReLU.

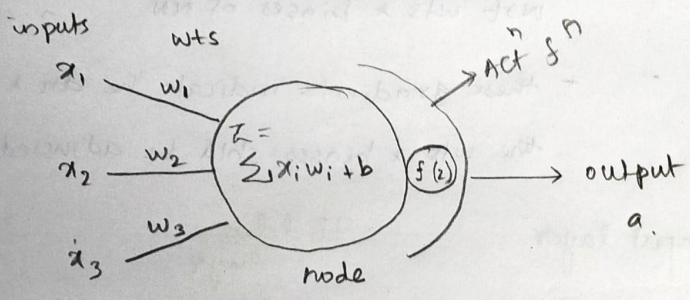
- An AF decides whether a neuron shld be activated / not. this means that it'll decide whether the neuron's input to the network is imp / not in the process of pred^n using math. operations.

- to transform the summed wt input from node into an output value to be fed to the next hidden layer or as output.

NN ≈ brain ⇒ node = neuron (that receives a set of input signals - external stimuli)

⇒ brain based on ↓ ⇒ processes them & decides whether neuron shld be act^d / fired or not.

- AF also called as transfer f^n



- input is fed into input layer, the neurons perform a linear trans^n on this input using wts & biases.

then AF is applied $Y = Act^n (\Sigma (wt * input) + bias)$

- output of AF moves to hidden layer and process is repeated
  ↳ forward movement ⇒ forward propogation.

- what if output generated is far away from actual value
  ↳ backwar. prop.

using this error is cal., based on this wts & biases of ~~from~~ updated

- A network w/o AF → will perform only linear transf of the wts & biases (input)

- linear transf → simpler but NN would be less powerful & not be able to learn complex patterns from the data

- NN w/o AF in DL is essential just as linear Reg. Model

∴ we use non-linearity fn for non-linear transf with the help of AF

- AF in NN → wt sum of input is transformed into an output from a node or nodes in a layer of the network.

- If output range of AF is limited → squashing fn

Many AFs are non-linear → non linearity in the layers of network design

- choice of AF → large impact on capability & performance of NN & diff AF fns may be used in diff parts of model

- may use within/after internal processing of each node in network

- Network layers — Input = raw input from domain
  - Hidden = input from another layer & pass it to out
  - output = makes pred^n.
    - ↳ diff AF & depends on the pred^n that model is making.

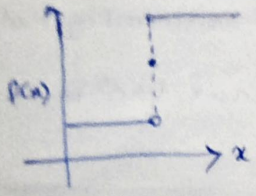
used AF

- AF ⇒ differentiable
  - ⇒ 1° order can be calculated for given input value.
  - ⇒ req given that NN are typically trained for BP of error algo that req the derivate of pred^n error in order to update the wts of the model.

* Types of AF
  - Binary step AF
  - Linear AF
  - Non-linear AF
    - tan-h
    - Sigmoid
    - ReLU
    - Softmax

① Binary step fⁿ

- depends on λ value that decides whether a neuron should be activated / not.
- input - fed to AF - compared w/ λ' if input > λ → ✓
  < λ ⇒ ✗
  its output isn't passed on to nxt HL

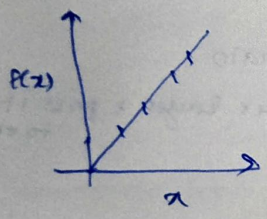


$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

limitations :-

- can't provide multi-value outputs (Multi-class classification problem x)
- gradient of step fⁿ = 0
  which causes a hindrance in BP process

② Linear / Identity AF



x is input
f(x) → output
f(x) = x [if x = 2 then y = 2]

- also known as "no activation fⁿ" or "identity fⁿ"
- AF α input
- fⁿ doesn't do anything, just simply spits out the value it was given.

limitations :-

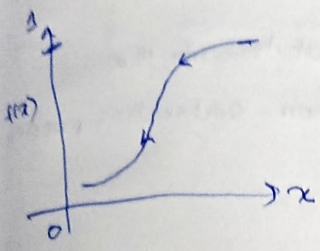
- not possible to use bp as the derivate of fⁿ is a const & has no. relation to the input x.
- All layers of NN will collapse into one if a linear AF fⁿ is used.

will be a linear fn of 1st layer. So, Linear AF turns NN to a layer (grey)

(B) Non-linear Activation fn :-

- they solve linear AF problems.
- Allow bp bcz now derivative fn is rel to input & its possible to go back and understand which wts in input neurons can provide a better pred^n.
- Multi layers of neurons as output would be non-linear comb^n of inputs passed from multi-layers.
- any output can be represented as fn computation in a NN.

(i) Sigmoid



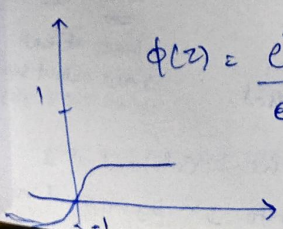
$$\phi(z) = \frac{1}{1+e^{-z}}$$

- Commonly used for models where we've to predict the prob. as an output.
- Since prob exists only blw 0 & 1, sigmoid is the ryt choice.
- fn is diff^{le} & provides a smooth gradient i.e., preventing jumps in output values.
  ↳ rep by S-shape of sigmoid fn.

limitations :-

- δ value $\longrightarrow 0$, network ceases to learn & suffers from vanishing δ prob.
- output of logistic fn is n't symmetric arnd 0, So the output of all neurons will be of same sign (0 to 1) ⇒ this makes training unstable and difficult.

(ii) tan-h



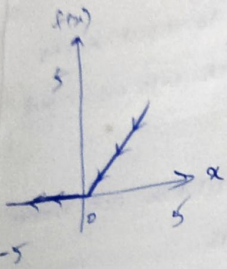
$$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

- Same s-shape but values [-1 & 1]
- it is zero centered, so we can easily map the outputs strongly -ve, neutral, strongly +ve
- used in HL as values [-1, 1]

∴ the mean for AL comes out to be 0 or close to it.

- centering data makes learning much easier

(iii) ReLU (Rectified Linear Unit)



$f(x) \Rightarrow \max(0, x)$

i.e., $x > 0 \rightarrow x$

$x \leq 0 \rightarrow 0$

- ReLU has a derivative fn & allows bp while simultaneously making the computation efficient.

- doesn't activate all neurons at the same time.

- neurons will only be deactd if the output of linear trans < 0

Adv :-

- only certain is actd, ReLU is far more computationally ~~activated~~ eff compared to Sigmoid, tan-h

- accelerates overall convergence of S descent towards the global min of fn due to its linear x non-saturating property

Disadv :- The dying ReLU problem.

(iv) Softmax AF

$$\text{Softmax}(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{k} K} \quad \text{for } j = 1, 2 \ldots K$$

- able to handle multiple classes.
- Sigmoid's prob by a sigmoid are independent ∴, not constrained to sum to 1
- Softmax input : -0.5, 1.2, -0.1, 2.4
  output : 0.04, 0.21, 0.05, 0.70

why? coz they look at each raw output separately

- whereas, Softmax's output are interrelated.

the sum will be always 1 ⇒ ↑se the likelihood of 1 class, other has to ↓se by an equal amt

* Optimization :- used for reducing the loss fn.

Algo :- GD
MBGD
RMSprop.

* Gradient Descent :-

- generalization of the derivative to the multivariate fn.
- descent ⇒ process of moving downward in fn's value i.e., to find min of a cost fn.
- local slope of fn.

Batch GD :- computes δ's and updates model parameters using the entire dataset at each step. → smooth/precise convergence but req significant computational power and memory.

Mini-Batch GD :- balances the trade-offs b/w BGD and Stochastic GD (SGD)

- updates model parameters using small subsets of data
- which in turn ⇒ faster training & better use of computational resources than BGD. (more stable convergence but less δ-variance than SGD)
- hybrid approach → ↑ly eff & widely used tech; esp w DL for large datasets.

BGP	SGD	MBGD
- entire dataset for updation	- single observation for updation	- subset of data for updation
- Cost fn reduces smoothly	- lots of variations in cost fn	- smoother cost fn as compared to SGD
- Computation cost is very high	- Computation time is more	- both are less time → SGD cost → BGD.

* RMS Prop :-

- we use adaptive learning rates.
- instead of a fixed η, learning rate changes based on past δ's.

* Bias-Variance Trade off :-

bias :- error from overly simple assumptions.
(model too simple → underfitting)

var :- error from model being too sensitive to training data noise.
(model too complex → overfitting).

Trade-off :- low bias → flexible model, ↑ risk of var.
low var → rigid model, risk of ↑ bias

So, we need to find a "sweet spot" where both are balanced → best generalization.

Overfitting :- model learns training data too well, including noise & irrelevant patterns.

- ↑ training accuracy but poor test accuracy.
- coz of too many parameters, insufficient data and lack of regularization.

* Regularization :-

- techniques to prevent overfitting by penalizing model complexity

ex :- L1 regularization (Lasso) → adds penalty ∝ |wts|
encourages sparsity scattered

L2 " (Ridge) → " ∝ |wts|²
reduces large coeffs.

elastic net → comb" of L1 & L2

- effect : shrinks wts → reduces var at the cost of slightly ↑er bias
→ improves generalization

* Generalization :-

- ability to perform well on unseen data
- opp of overfitting
- Achieved by balancing bias-var, using reg", dropout & choosing good inductive bias

* Inductive Bias :-
- set of assumptions a learning algo uses to make pred^{n} on unseen data
- W/o IB, learning from finite data is impossible.

ex:- LR assumes linear rel^{n}
Decision trees assumes data can be split hierarchically.
NN → complex patterns can be rep by layers of fn.

* Dropout :-
- A reg^{n} tech used in NN ; randomly drops out (sets to 0) a subset of neurons during training
- prevents co-adaptation of neurons → reduces var
- acts like an ensemble (group) of many smaller networks
↳ genaral^{n} improve.

* Probabilistic Neural Network (PNN)

- used for classification & pattern recognition tasks
- Bayesian Stats & estimates PDFs for each classes, rather than learnin' wts thru iterative bp.
- 4 layer FNN (one dir w/o any cycles) In → out

4 Layers :- (2 feature : Apple & orange)

(i) Input layer :- just receives feature vector (that needs to be classified) & feed to pattern layer
- No calculation
- distrib. the input feature to every neuron in the next layer
if fruit w wt = 150g, Color = 0.8 ↳ give to PL
dim^{n} of layer = 2

(ii) Pattern layer/Hidden layer :-
- memorizes the entire dataset & measure how similar the new inputs is to every single training example.
- there is 1 neuron for each observation (100 fruits = 100 neurons)
- each neuron is associated w. both a specific training vector & its known class
(1 neuron rep a specific apple from T.D)
- calculates euclidean dist b/w input vector (X) & its own stored training vector (x)
- then feeds the dist into Radial Basis fn (RBF) or Gaussian fn
- this fn outputs a value that rep the 111^{th} or act^{n}.
- closer the input is to neuron's stored vector, 1^{er} the output

Ex: input [150, 0.8] compared w neuron w [145, 0.75] apple ↑ value
orange ↓ value

(iii) Summation layer:-

- avg the votes from PL for each class
- 1 neuron for each class → collects the output from all pattern neurons that belong to its class
- Sums these outputs & avg them by total no. of patterns
- avoids unfair due to more training samples.

(iv) Output layer :-

- make final classification decision
- Choose the largest summation layer neuron.

3. apple = 0.85
3. orange = 0.12

→ Parzen window est^n → PNN approx^s the parent PDF for each class

adds the value and approx to a prob hill

(non-parametric tool)
↳ doesn't assume anything

→ don't use bp for training

NeuPy → python lib for Prototyping & building ANN

neupy.algorithms.PNN

check front

RNN ⇒ Recurrent NN

- process sequential data (txt, speech / time series data) by maintaining an internal "memory" of past info.
- unlike FNN, this uses past info from 1 step to next to understand the context & dependencies within a sequence

* Hopfield Network

brain can identify something, why? this do

- RNN network invented by John Hopefield.
- uses CAM (Content Addressable Memory), stores & retrieve patterns based on partial or noisy inputs, both completes & find errors.
- single layer of interconnected neurons w/ symm wts, where the network's dynamics, driven by an energy fn, cause it to converge to stored mem. states.
- binary neurons [either -1 or 1 / 0 or 1]

$w_{ij} = w_{ji}$

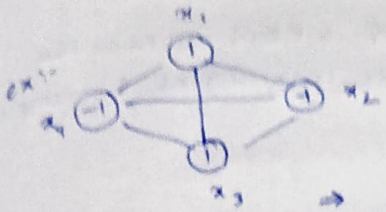
every neuron connected to others

$net = \sum_j (w_{ij} \times s_j)$ sum of all states × connection wts.

$E = -\frac{1}{2} \sum_i \sum_j w_{ij} \times s_i \times s_j$

if $net_i \geq 0 \rightarrow s = +1$
else $net < 0 \rightarrow s = -1$

set wt ⇒ wt > 0 → same state
wt < 0 → diff.



Given M target N-S = $\{x^{(1)} \dots x^{(m)}\}$

$$w_{ij} = \frac{1}{M} \sum_{s=1}^{M} x_i^{(s)} x$$

$$W = \begin{bmatrix} 0 & -1 & 1 & -1 \\ -1 & 0 & -1 & 1 \\ 1 & -1 & 0 & -1 \\ -1 & 1 & -1 & 0 \end{bmatrix}$$

[1, -1, 1, -1]
[1, 1, -1, -1]
⇒ 1×-1 = -1
1×1 = 1
$\frac{1}{2}[1-1] = 0$ (w12)

Boltzmann Machine :-

- stochastic RNN that learns prob Dist. over its inputs.
- energy-based generative model that uses prob. approach to learn patterns in data by min energy fn
- consists of interconnected visible & hidden neurons learns PDFs & lest energy state.
- due to its slow ~~ener~~ learning process, a restricted version called RBM limits connections b/w layers was developed to improve eff.
- neuron constitute a recurrent structure & they operate in binary manner ⇒ On-state ⇒ +1
  off-state ⇒ -1

Visible nodes :- receive input data from external world
Hidden nodes :- Capture internal features & representations
All to All conn^ns ⇒ every neuron is connected to every other neuron
V-V, V-H, H-H
Boltzmann Distri :- ⟶ energy.

- energy fn ⇒ $E = \frac{-1}{2} \sum_i \sum_j w_{ij} \times s_i \times s_j$ (wt b/w i & j; n·s)
  ⊖ → low energy = more stable state
  i ≠ j ↳ none has self feedback

- Chooses a random neuron 'k'
  flips the state from x_k to x_{-k} ⇒ Check the temperature T

$$P(x_k \to x_{-k}) = \frac{1}{1 + e^{-\Delta E_k / T}} \quad \text{(sigmoid)}$$

ΔE_k → energy change
T → Pseudo temp

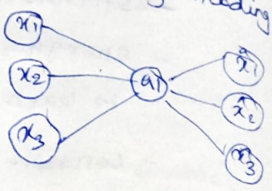
↑ temp → Large explores widely
↓ temp → 0 ⇒ greedy
$P(\text{flip}) = 1$ if $\Delta E_k > 0$ ⇒ flip
$P(\text{flip}) \to 0$ if $\Delta E_k < 0$ ⇒ don't
Shld reach ⇒ thermal eq

- energy based Model (↓ energy = more stable)
- sym. connections ⇒ $w_{12} = w_{21}$
- Stochastic decisions ⇒ prob. decisions to be ON/OFF
- learning thru energy Minimization ⇒ learns by adjusting its internal wts & biases → ↓ overall energy.

* Deep BM
  - deep NN based on BM
  - Stacked RBM, where output of 1 RBM serves as the input for the nxt, allowing for learning of hly complex data representations.

* **AutoEncoders in DL**

- unsupervised NN that learn to reconstruct their input data
- compression algo, learning eff data representations by encoding and decoding the data.
- ignore noise in data.



Encoder :-
- a module that compresses the train validated test ~~set~~ input data into a encoded rep. i.e., several orders of mag smaller than the input data.

ex:- Img → convolutional layers + pooling layers. 1^{er} dim → $\downarrow^{er}$ dim

- set of convolutional blocks + pooling layers that compress the input to model into compact section called bottleneck

Bottleneck :-
- latent space
- contains compressed data (knowledge rep)
- captures only most imp features of data to allow successful reconstruction.

- a bottleneck as a compressed rep of an input further prevents the NN from memorizing the input & overfitting on the data.

- Smaller the BN, lower the risk of overfitting.
- However, v. small BN would restrict the amt of info stored, which ↑es the chances of imp info slipping out thru the pooling layers of encoder.

Decoder :-
- helps the network decompress the knowledge rep & reconstructs the data from its encoded form.
- Output is then compared w ground truth
- set of unsampling & convolutional blocks that reconstruct

Applns :-

→ Dim Redⁿ : easy analysis & visualization

→ Feature Extraction

→ Noise Removal : remove noise from data.

→ Anomaly detection : reconstruct → can identify anomalies as input that they can't reconstruct accurately.

→ Generative Models : generate new data

Hyperparameters b4 training autoencoder :-

→ Code size / BN size : most imp
it decides how much data to be compressed
Small : strong ; Large : Weak
risk of losing imp info
risk of overfitting

→ No. of Layers : depth of encoder & decoder
↑er depth ↑es model complexity
↓er → faster to process

→ No. of nodes per layer :- wts we use per layer
no. of nodes ↓es w subsequent layer
as input becomes smaller

→ Reconstruction loss :- loss f^n we use ↑ly dependent on type of input & output we want the auto to adapt to.
image ⇒ MSE loss & L1 loss
output in range [0,1] ⇒ MNIST,
Binary cross entropy.

Types :-

→ Std AE :- basic AE that learns a compressed rep of input data.

→ De-noising AE :- Trained to reconstruct the original data from noisy data.

→ Variational AE :- learns prob-rep of data

→ Convolutional AE : imgs & other spatial data.

* Probabilistic Graphical Models :-

linear chain :- each node depends on its ~~prev one~~ neighbour :- sequence

cyber :- sys logs → user act → packets / time
for anamoly detection.
open → read → write → close → Normal
open → read → exec → delete → Malicious behav.

Partition f^n (Z) :- * normalization const used in prob. models
prob sum = 1

cyber :-
eval prob of given attack pattern, parition f^n
helps in normalize over all possible sys states
helps in ranking diff attack hypotheses.

Malwar detection model ⇒ 1

Markov Random field :- nodes & their dependencies w/o assuming direction
spatial / relational dependencies

cyber :- can detect inter-dependencies among devices / hosts in a network

useful in DIDS (distributed Intrusion Detection systems)

Combined detects { Node A → abnormal traffic
B → logs unknown outbound connection
C → shows privilege escalation.

Belief Propogation :- (BP)
Algorithm used in PGMs to compute marginal Prob.

cyber :-
helps in propogating across the network of events, one event is likely to happen.

ex :- phishing email leading to cred compromise (A) triggering lateral movement (B)
BP can propagate the belief from A to B, & detection confidence.

Training Conditional Random Fields :-
discriminative probabilistic model structured in linear chain / graph

- involves max the conditional likelihood of labeled sequence.

cyber :-

CRFs used to label seq. network traffic flows / sys logs.

they learn dependencies b/w features & labels.

open → chmod → sus.

Hidden Markov Model (HMM) :- hidden states (normal, attack) and observable outputs (log entries / pkt features)

cyber :- used for behav modelling, intrusion detection.

HS :- {normal, exploit, data exfiltration}

Obs :- {port scanning, unusal login, large file transfer}.

Entropy :- measures uncertainity / randomness of data.

cyber :- ↑ entropy in files ⇒ likely encrypted / malware

↓ in Net traffic ⇒ normal comm^n

Sudden change ⇒ potential intrusion.