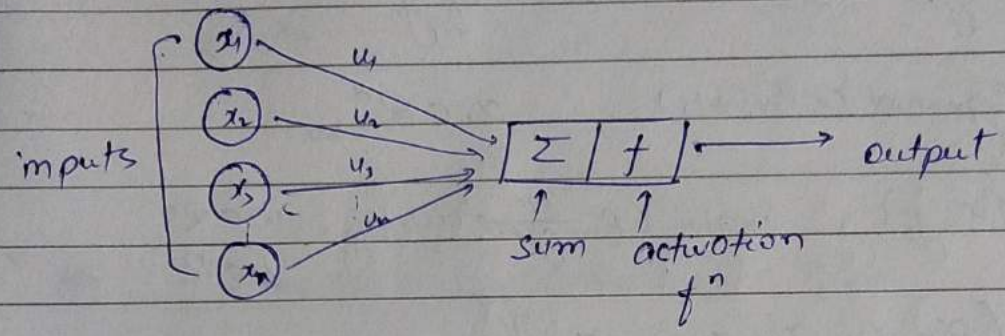
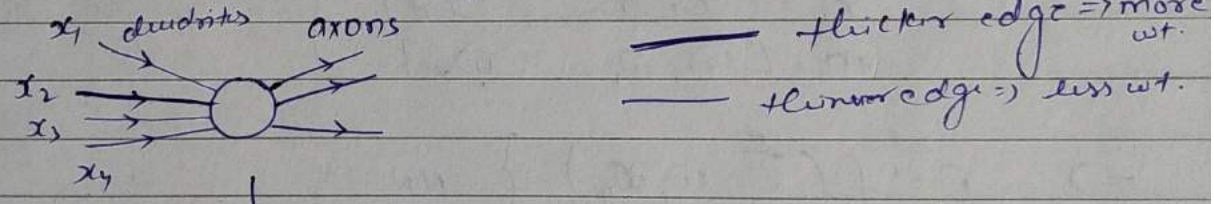
DEEP LEARNING

✱ History of Neural networks :-



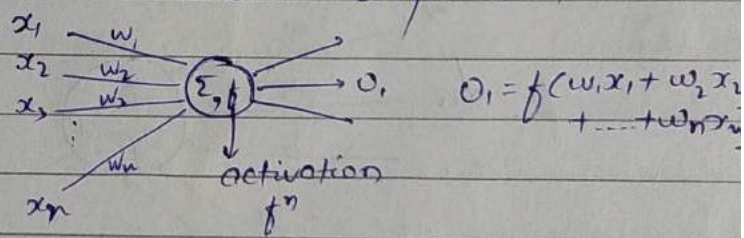
✱ How Biological Neurons work?

Simplified view of neuron.



Youtube: Programming Cradle

→ Mathematical representation



$$O_1 = f(w_1x_1 + w_2x_2 + \dots + w_nx_n)$$

$$O = f\left(\sum_{i=1}^{n} w_i x_i\right)$$

O: output, f: activation f^n, w_i: wt., x_i: input

✱ Growth of biological neural networks.

→ formation of new branches b/w neurons

→ thickness of branch depends on how much important it is.

⊛ Diagrammatic Representation: Logistic Regression and perceptron (Single layered perceptron)

LR: $x_i \rightarrow \hat{y}_i \rightarrow$ predicted value of y_i

$\hat{y}_i = \text{Sigmoid}(w^T x_i + b)$ $\quad x_i \in R^d$

$D = \{x_i, y_i\}$ train LR $\xrightarrow{\text{we get}}$ w, b $\quad w \in R^d$, $b \in R$

$x_i = [x_{i1}, x_{i2}, x_{i3}, x_{i4} \ldots x_{id}]$

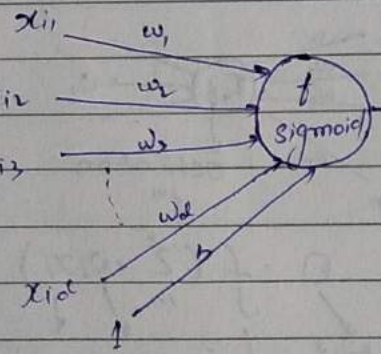
$$\Rightarrow \hat{y}_i = \text{Sigmoid}\left(\sum_{j=1}^{d} w_j x_{ij} + b\right)$$

$x_i = [x_{i1}, x_{i2}, \ldots x_{id}]$

$w_i = [w_1, w_2 \ldots w_d]$

} LR

$$\Rightarrow O = f\left(\sum_{j=1}^{d} w_j x_{ij}\right)$$ } NN



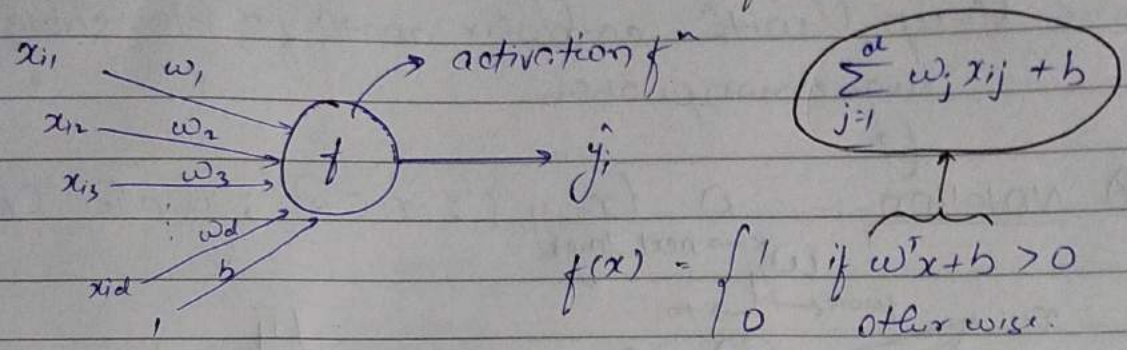
$\hat{y}_i = f(w_1 x_{i1} + w_2 x_{i2} + \ldots + w_d x_{id} + 1 * b)$

$\Rightarrow \hat{y}_i = f(w^T x_i + b)$

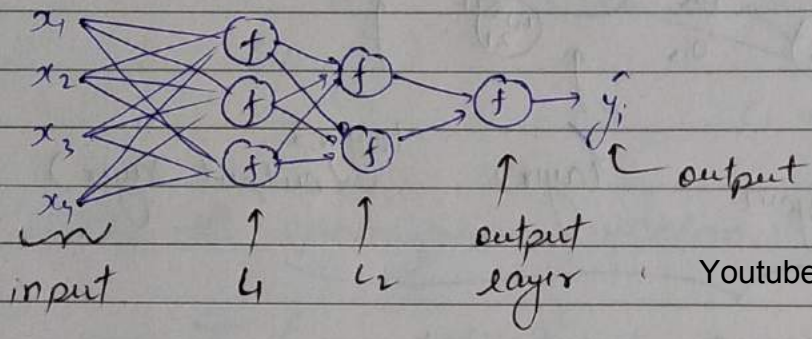
⇒ Train a neural network means finding weights on edges / vertices

⇒ Only diff^n b/w perceptron and LR is the loss f^n in LR we do squashing using Sigmoid f^n whereas in perceptron we don't do squashing.

Perceptron (1957) → linear classifier



Multi-layered Perceptron (MLP)



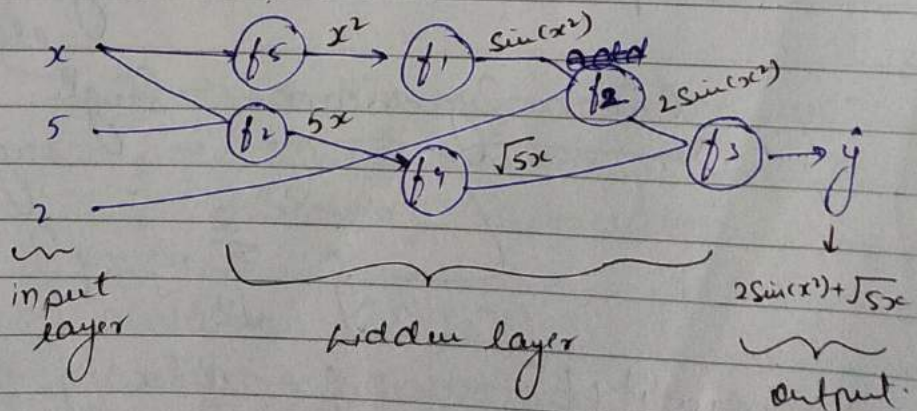
Q) Why should we care about MLP?

→ Biological inspiration : biological neurons are connected in multiple layers (millions)

→ Mathematical argument :-

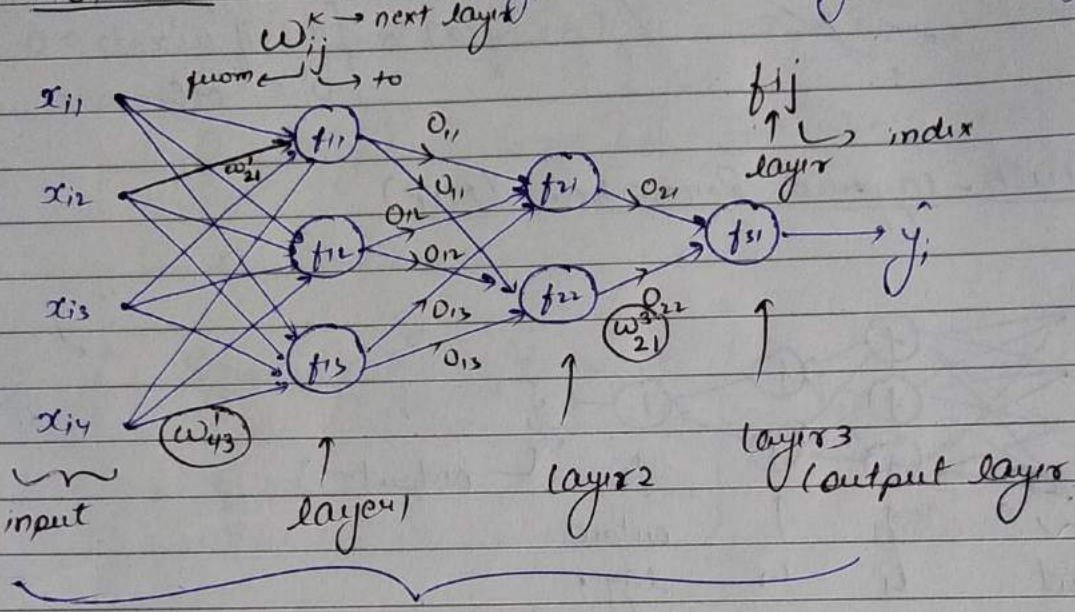
example :- $2 * \sin(x^2) + \sqrt{x5}$

$f_1 = \sin()$
$f_2 = mul()$
$f_3 = add()$
$f_4 = sqrt()$
$f_5 = square()$



* By using multi layered structure we can come up with complex math f^n to solve your regression problem.

* Notation :- $D = \{x_i, y_i\}$; $x_i \in R^4$; $y_i \in R$ (Regression)



fully connected NN

Youtube: Programming Cradle

* Train a Single Neuron model :-

↳ (find the best edge-weights using D_{tr})

Perceptron and LR → Single Neuron models for classif^n

Linear Regression → Single Neuron models for regression

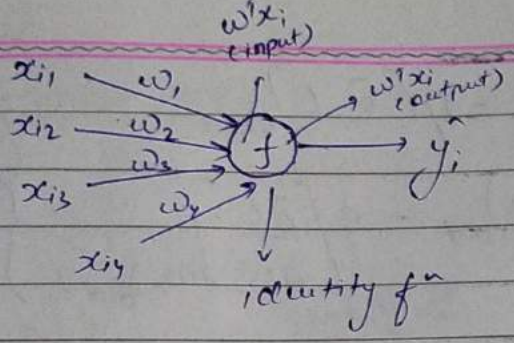
$$\hat{y}_i = \sum_{j=1}^{d} w_j x_{ij}$$

$$\hat{y}_i = w^T x_i$$

} linear reg. opt^n

identity f^n

$f(x) = x$



[in logistic reg^n f - sigmoid f^n]

Linear Regression optimizn :-

Define
① loss f^n $\min_{w_i} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + reg$ → $\|w\|_1$ or $\|w\|_2^2$

n → no. of points.

$\hat{y}_i = w^T x_i$

$L_i = (y_i - \hat{y}_i)^2$

L → Sum of all the points
L_i → for a particular point.

② Write the optimization problem.

$$\min_{w_i} \sum_{i=1}^{n} (y_i - w^T x_i)^2 + reg \quad \text{— ①}$$

$\hat{y}_i = f(w^T x_i)$
↳ in linear reg^n f → identity f^n

Youtube: Programming Cradle

⇒ writing eq^n ① in genral form.

$$w^* = \underset{w_i}{\operatorname{argmin}} \sum_{i=1}^{n} (y_i - f(w^T x_i))^2 + reg.$$

↳ f → identity f^n for linear reg^n
f → logistic / sigmoid f^n for LR
f → threshold f^n ← perceptron.

③ Solve the optimizn problem (SGD)
a) initialize of w_i's → random.
b) $\nabla_w L = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \vdots \\ \frac{\partial L}{\partial w_n} \end{bmatrix}$ $w \in R^n$ $x_i \in R^n$

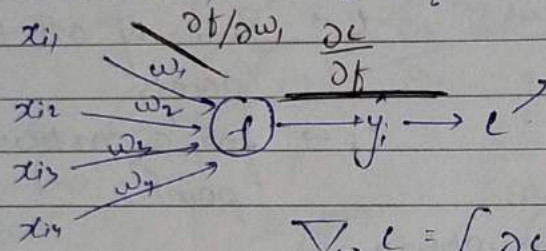
ⓒ $\omega_{new} = \omega_{old} - \eta(\nabla_\omega L)_{\omega_{old}}$

$(\omega_i)_{new} = (\omega_i)_{old} - \eta\left[\frac{\partial L}{\partial \omega_i}\right]_{(\omega_i)_{old}}$ } updating weights. update eqⁿ

GD: $\nabla_\omega L \rightarrow x_i$'s and y_i's

SGD: $\nabla_\omega L \approx$ one pt $\{x_i, y_i\} \leftarrow$ small batch of pts $\leftarrow$ batch SGD

Q) How to compute $\nabla_\omega L$?



$$\nabla_\omega L = \left[\frac{\partial L}{\partial \omega_1}, \frac{\partial L}{\partial \omega_2}, \frac{\partial L}{\partial \omega_3}, \frac{\partial L}{\partial \omega_4}\right]^T$$

$$\frac{\partial L_i}{\partial \omega_1} = \frac{\partial L_i}{\partial f} \cdot \frac{\partial f}{\partial \omega_1} \longrightarrow \text{chain rule of diffⁿ}$$

$$\Rightarrow L = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + (\text{eq.}) \rightarrow \text{ignor for now}$$

$$= \sum_{i=1}^{n} (y_i - f(w^T x_i))^2$$

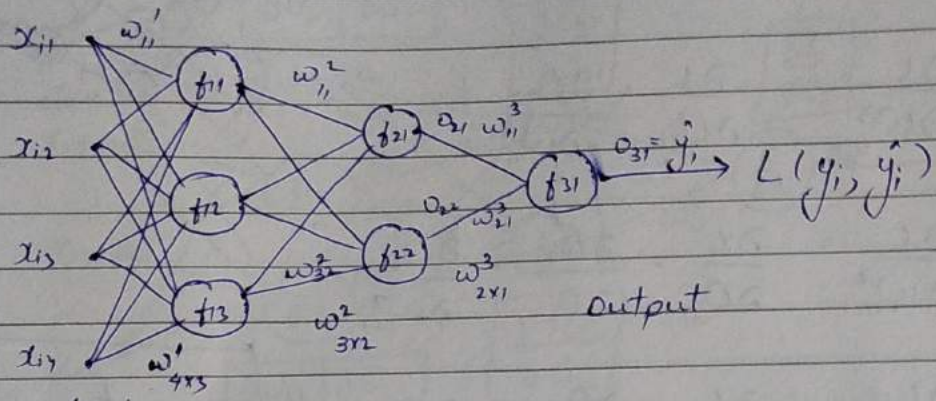
$$\frac{\partial L_i}{\partial f(w^T x_i)} = \frac{\partial L_i}{\partial f} = -\sum_{i=1}^{n} 2(y_i - f(w^T x_i))$$

$$\frac{\partial L_i}{\partial \omega_i} = -2x_{ii}(y_i - \hat{y}_i) \quad f \rightarrow \text{identity fⁿ } f(x) = x$$

$$f(w^T x_i) = w^T x_i$$

$$\frac{\partial L}{\partial \omega_1} = \sum_{i=1}^{n} -2x_i(y_i - \hat{y}_i) = -\sum_{i=1}^{n} 2x_i(y_i - f(w^T x_i))$$

✱ Training MLP



Training means: finding $\underbrace{\omega'_{4\times2}, \omega^2_{3\times2}, \omega^3_{2\times1}}_{\text{20 wts.}}$

① Define loss fⁿ

$$\Rightarrow L = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \text{reg} \rightarrow \sum_{ij,k} (\omega^k_{ij})^2 \quad \text{← } L_2$$

or

$$\sum_{ij,k} |\omega^k_{ij}| \quad \text{← } L_1$$

$$\Rightarrow L_i = (y_i - \hat{y}_i)^2 \Rightarrow L = \sum_{i=1}^{n} L_i + \text{reg.}$$

② Optimizⁿ :-

$$\min_{\omega', \omega^2, \omega^3} L \rightarrow \text{sq. loss + regul}^n$$

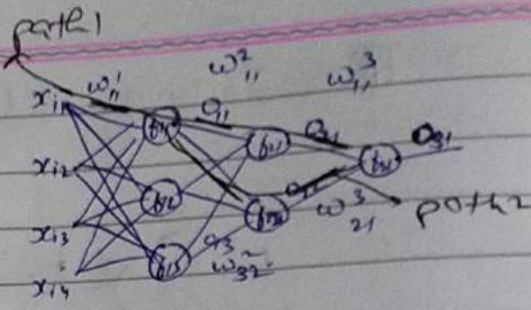
$$\boxed{\min_{\omega^k_{ij}} L}$$

③ SGD or GD

a) initializⁿ of ω^k_{ij} randomly.

update eqⁿ → b) $(\omega^k_{ij})_{\text{new}} = (\omega^k_{ij})_{\text{old}} - \eta \frac{\partial L}{\partial \omega^k_{ij}}$ ← learning rate

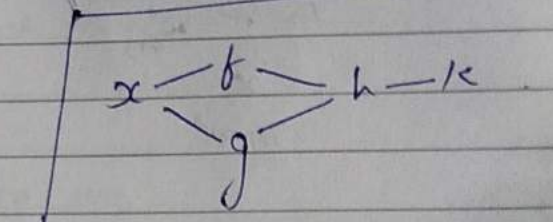
c) perform update till convergence.



$$\omega^3 \begin{cases} \frac{\partial L}{\partial \omega^3_{21}} = \frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial \omega^3_{21}} \\ \frac{\partial L}{\partial \omega^3_{11}} = \frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial \omega^3_{11}} \end{cases}$$

$$\omega^2 \begin{cases} \frac{\partial L}{\partial \omega^2_{11}} = \frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial \omega^2_{11}} \\ \frac{\partial L}{\partial \omega^2_{21}} = \frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial \omega^2_{21}} \\ \vdots \\ \frac{\partial L}{\partial \omega^2_{32}} = \frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial O_{22}} \cdot \frac{\partial O_{22}}{\partial \omega^2_{32}} \end{cases}$$

$$\omega^1 \quad \partial\omega^1_{11} = \frac{\partial L}{\partial O_{31}} \cdot \Big(\underbrace{\frac{\partial O_{31}}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial O_{11}} \cdot \frac{\partial O_{11}}{\partial \omega^1_{11}}}_{\text{path 1}} + \underbrace{\frac{\partial O_{31}}{\partial O_{22}} \cdot \frac{\partial O_{22}}{\partial O_{11}} \cdot \frac{\partial O_{11}}{\partial \omega^1_{11}}}_{\text{path 2}} \Big)$$



$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial x}$$

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial f} \cdot \frac{\partial f}{\partial x} + \frac{\partial h}{\partial g} \cdot \frac{\partial g}{\partial x}$$

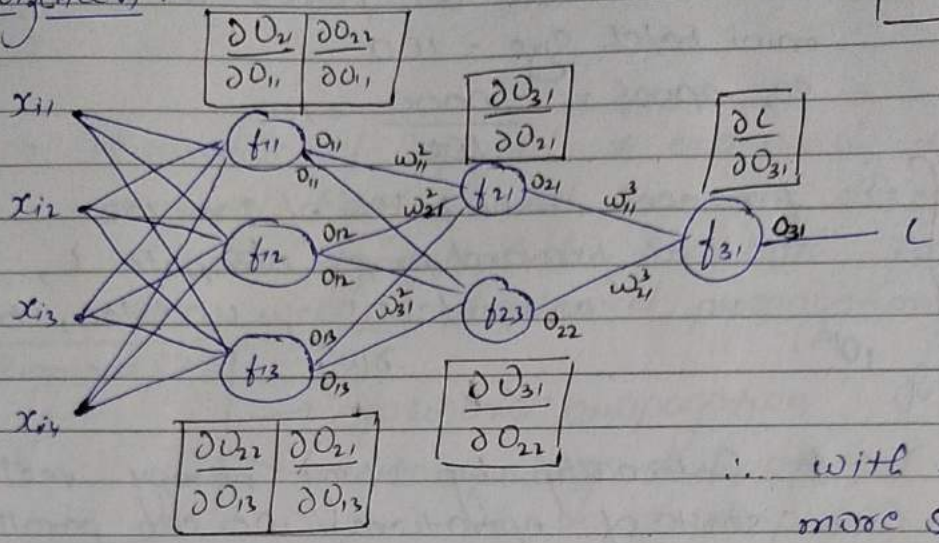
$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial h} \Big(\frac{\partial h}{\partial f} \cdot \frac{\partial f}{\partial x} + \frac{\partial h}{\partial g} \cdot \frac{\partial g}{\partial x} \Big)$$

⊛ Training MLP memoization

memoization :- if there is any operation that is used many times repeatedly, it is a good idea to compute it once and store it for future use.

Epoch :- no. of times whole data is passed through the network

Memoization :-



∴ with slightly more space, speed can be ↑ drastically.

* Backpropagation :- Chain rule + memoization

$D = \{x_i, y_i\}$

① initialize w^k_{ij}'s

② for each x_i in D

a) pass x_i forward through the nw :- forward prop

b) Compute $L(\hat{y}_i, y_i)$

c) Compute all the derivatives using chain-rule and memoization

d) update weights from end of the network to start : back propagation

③ Repeat step 2 till convergence

- Instead of sending one point at a time to the network (SGD) we can send batch of points (mini batch SGD)
- Keeping all the datapoints in RAM and computing derivatives (∂) is hard (next to impossible if data is huge) } GD
- To over come above problem mini-batch based back prop is used most of the times

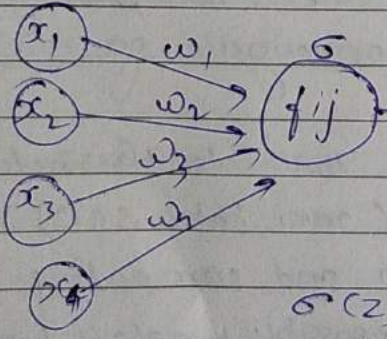
Let there are 10K datapoints in D and
mini batch size = 100
so, epoch = $\frac{10000}{100} = 100$

we have to run this all 100 times instead of 10K

=> for each mini-batch of size 100
→ forward prop, compute L,
compute $\frac{\partial L}{\partial w_{ij}^k}$, update, back prop.

* Internally by using numpy vectorizn sort of operations, we can parallely send multiple inputs to the n/w. think of them as a matrix with datapoints one stacked on top of another. While calc. the gradient, gradient for all the pts. is calculated and avg. value is taken which is used for updating the wts.

* Activation function :- fij (Sigmoid, tanh) → popularity (80, 90)
↳ activation f^n should be differentiable and easy to differentiate.



$z = \sum_i w_i x_i = w^T x$

$w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}$

$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$

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

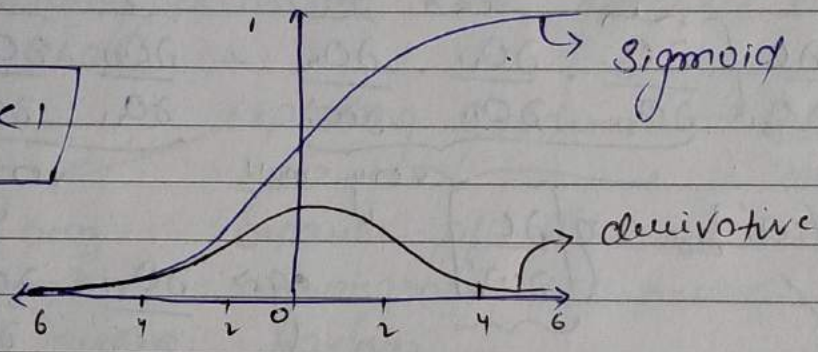
$$\frac{\partial \sigma}{\partial z} = \sigma(z)(1-\sigma(z))$$

→ derivative of Sigmoid can be expressed in terms of sigmoid itself and hence we can use sigmoid code for both forward as well as backward propagation.

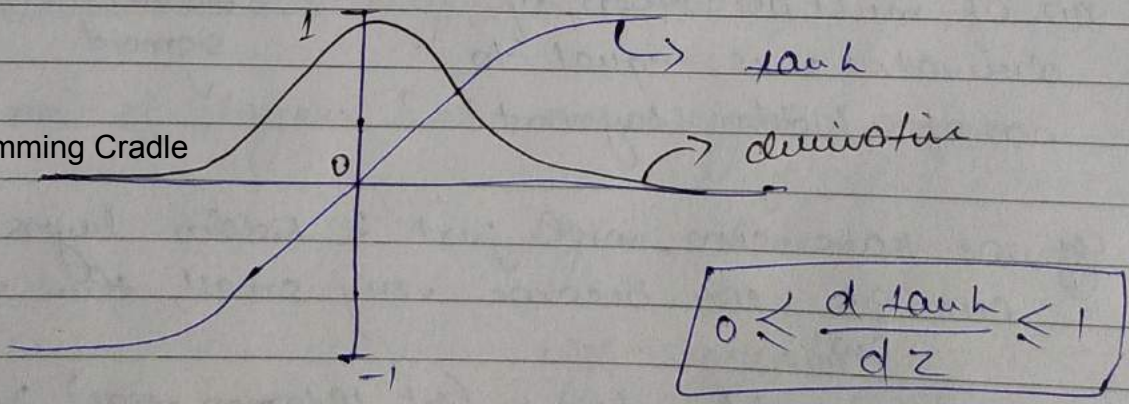
```
Sigmoid(z)
{
}
```

→ forward propagation
→ computing derivatives (done in back prop)

$$0 \leq \frac{d\sigma}{dz} \leq 1$$



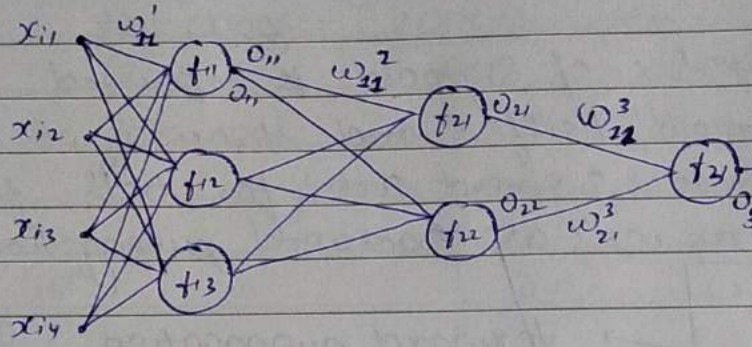
* $\tanh = \frac{e^z - e^{-z}}{e^z + e^{-z}} = a \Rightarrow \frac{da}{dz} = 1 - a^2$



$$0 \leq \frac{d \tanh}{dz} \leq 1$$

Modern activation fn : ReLU

⊛ Vanishing gradients :- (Biggest problem in NN during 80's, 90's and 2000's)



→ As f_{ij} is sigmoid (most of the times), value lies b/w 0 and 1

$$\frac{\partial C}{\partial w'_{11}} = \frac{\partial C}{\partial o_{31}}\left[\underbrace{\frac{\partial o_{31}}{\partial o_{21}}\cdot\frac{\partial o_{21}}{\partial o_{11}}\cdot\frac{\partial o_{11}}{\partial w'_{11}}}_{\text{very small}} + \underbrace{\frac{\partial o_{31}}{\partial o_{22}}\cdot\frac{\partial o_{22}}{\partial o_{11}}\cdot\frac{\partial o_{11}}{\partial w'_{11}}}_{\text{very small.}}\right]$$

$(0<\frac{\partial}{\partial}<1)$

$$(w'_{11})_{new} = (w'_{11})_{old} - \eta\left[\frac{\partial C}{\partial w'_{11}}\right]$$

will be almost same

very small

eg.→ $\frac{\partial o_{31}}{\partial o_{21}}\cdot\frac{\partial o_{22}}{\partial o_{11}}\cdot\frac{\partial o_{11}}{\partial w'_{11}}$

$0.2 * 0.1 * 0.05 = \text{0.001}$ $\ll 1$

these are small (b/w 0 to 1) because of sigmoid

Youtube: Programming Cradle

⊛ no. of multiplication of derivatives is equal to no. of hidden layers + 1

⊛ If we have sum, with just 2 hidden layers (3 multip) our $\frac{\partial C}{\partial w'_{11}}$ can become very small, then with more hidden layers (let 10 to even more) $\frac{\partial C}{\partial w'_{11}}$ will tend to 0.

⊛ As $\frac{\partial C}{\partial w'_{11}}$ getting smaller and smaller it is called as Vanishing gradient

* Sigmoid / tanh → $0 < < 1$ → chain rule and multiplication
↓
$(w_{ij}^{k})_{new} = (w_{ij}^{k})_{old}$ ← vanishing gradient problem ← v.v small $\frac{\partial L}{\partial w_{ij}^{k}}$

* Because of above problem deep RNNs could not be trained.
* To solve above problem ReLU activation f^{n} was discovered.
* There is a problem known as Exploding Gradient it mainly occurs in RNN. We will see this in RNN chap.

Exploding Gradient → $\frac{\partial L}{\partial w_{ij}^{k}}$ → very large.

this happen bcoz ($\frac{\partial L}{\partial w_{ij}^{k}}$) components will be greater than 1 and when we multiply values greater than 1 it becomes v. large

because of this convergence will never happen.

* Bias - Variance Tradeoff :- (MLP)

* no. of layers ↑ => more weights / params
↓
Higher chance of overfitting
↓
High variance.

* no. of layers ↓ => less wts => high chance of underfit
↓
High bias

Youtube: Programming Cradle

* To get rid of overfitting and underfitting we can use L_1 or L_2 regul^n.
* To get rid of over/underfitting we fine tune hyperparam:
  1. α ($L = \sum_i loss_i + \alpha L_2$ reg)
     - $\alpha \uparrow \Rightarrow$ var $\downarrow$
  2. no. of layers :-
     - no. of layers $\uparrow \Rightarrow$ var $\uparrow$
* Deep Multi-layer perceptrons (1980s to 2010s)
  - failed during (80-2010) because
    - vanishing gradient
    - too little data :- beog of which overfitting is ~~possible~~ common
    - too little computation power. hence it took too much time to train.
* But by the end of 2010 → helped in modern DL
  1. we had lots of data (also labelled data like imagenet)
  2. computation power → powerful CPUs and/or GPUs
  3. New ideas and algorithms.
* classical ML — SVM, RF { 1st theory was developed then → experimental proofs }
* Modern DL → experiment first then find theoritical proofs.

Youtube: Programming Cradle

⊛ Dropout layers and Regularization

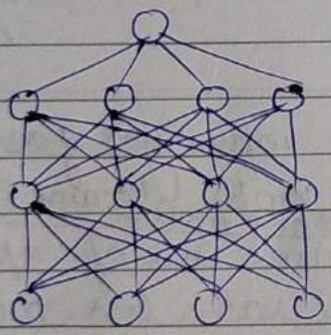
⊛ ⇒ It is easy to overfit in Deep NN because there are lots of weights.

⊛ One way to avoid overfitting is by using L_1 and L_2 regularization. Other way can be "dropout" layers.

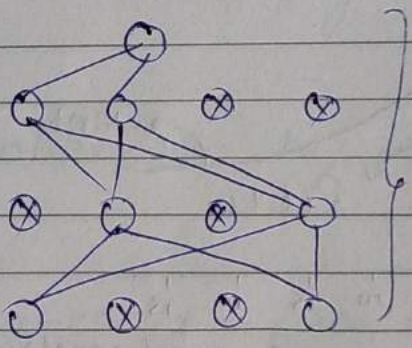
⊛ In Random Forest we use randomization of features to create regularization

- ⇒ in Random forest we create multiple fully grown trees on subset (randomly selected features) of features and then we take majority vote

⊛ <u>Dropout</u> ≈ random subset of features in RF

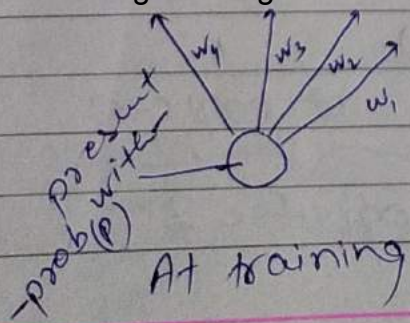


⇒ Standard NN

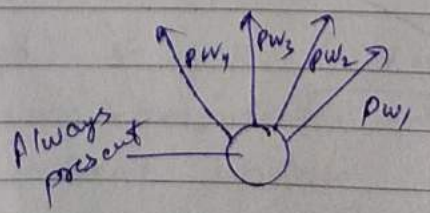


⇒ After applying dropout.

⊛ Dropout Rate :- $0 < p < 1$ — In a particular layer what percent of neurons / nodes will be inactive / dropped out.

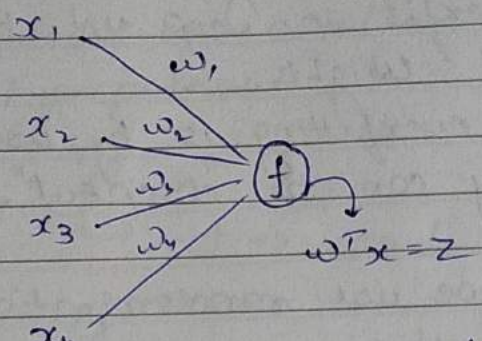


At training time



At test time

⊛ Rectified Linear Unites (ReLU)

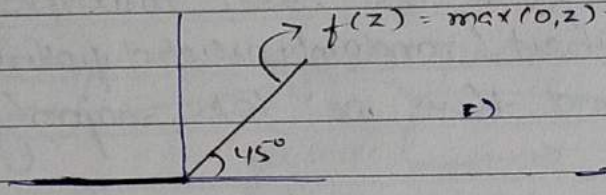


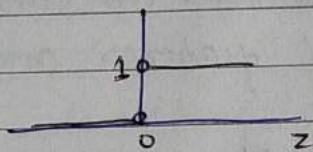
$f(z) = z^+ = \max(0, z)$

ReLU

not differentiable at zero.

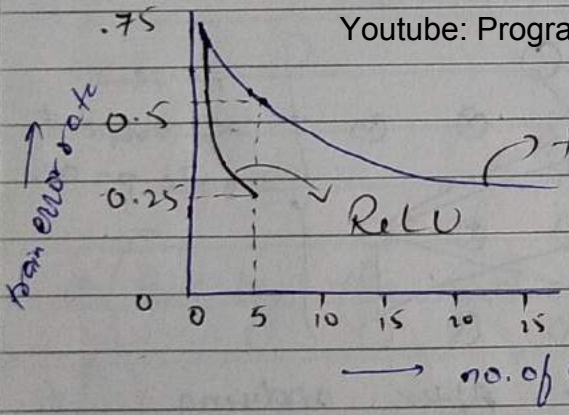
$f(z) = \max(0,z) = \begin{cases} 0 & \text{if } z \le 0 \\ z & \text{otherwise} \end{cases}$





bcoz of zero, dead activation problem may occur.

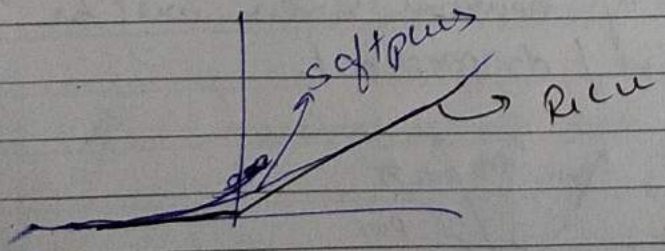
$\frac{df_{ReLU}}{dz} \in \{0, 1\}$ → no exploding + no vanishing problem



→ ReLU converges faster than tanh because ReLU don't have vanishing grad problem

→ for same no. of epochs ReLU has less error than tanh

⊛ Smooth approximation of rectifier $f(x) = \log(1 + \exp x)$



Softplus

$f'(x) = \frac{1}{1 + \exp(-x)}$

logistic fⁿ

* problem with ReLU :-
  when z is negative $f(z) = 0$ and $\frac{df}{dz} = 0$

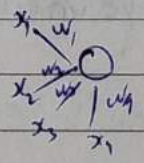
  and $\frac{\partial L}{\partial w_{ij}^k}$ will become 0 $\left(\frac{\partial L}{\partial w_{ij}^k} = \{ \text{multiplication of several intermediate derivatives} \} \right.$

  we know that $(w_{ij}^k)_{new} = (w_{ij}^k)_{old} - \eta \frac{\partial L}{\partial w_{ij}^k}$ (eqn ①)

  $1 \times 1 \times 1 \times \dots \times 0$ — even if 1 deriv is '0' whole mult. will become zero

  if in eqn ① $\frac{\partial L}{\partial w_{ij}^k} = 0$ convergence will not happen. → this problem is known as dead activation

  ⇒ what is z ?
  $z = w^T x$

  x_i → always normalized

* One soln for dead activation can be, we can use Leaky ReLU.

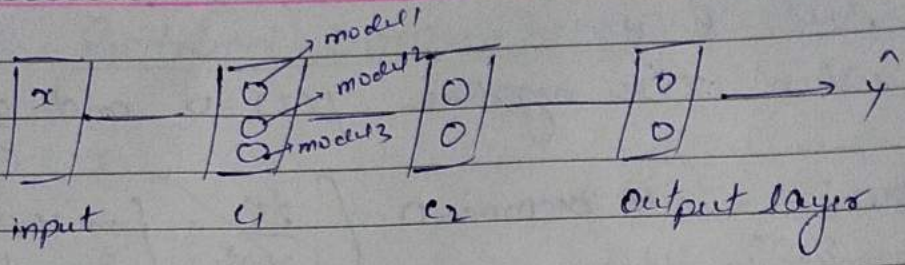
→ In Leaky ReLU when z is -ve we use some small value a which is b/w 0 and 1 instead of using '0' but this can again lead to vanishing prob.
→ a → hyperparam.

* Weight initialization :- (Deep MLP)

① init $w_{ij}^k = 0$ (or any no.) $\forall i, j, k$ } very bad idea beoz:

Symmetry but we want asymm. ←
- all neurons compute the same thing
- same gradient update

beoz of this → each node in our model / MLP isn't learning anything



* More different each models are, the better will be the performance. (same concept from RF and GBDT where, more diff" base models results in better performence)

② init w_{ij}^{k} = large -ve number $\forall i,j,k$

$w^T x = z$ = large-ve values $\Rightarrow f(z) = 0$

↑ normalized ↑ ReLU

Youtube: Programming Cradle

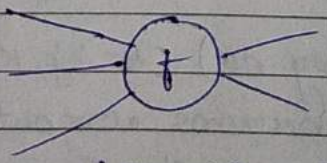
Solution :-

idea 1 :- -> weights should be init. with small no. (not very small)
-> not all zero.
-> good variance ⇒ (diff node/models)

$w_{ij}^{k} \sim N(0, \sigma)$ σ: small

↳ weights are normally distributed.

* Better init strategies :-
-> use fan-in and fan-out for init.

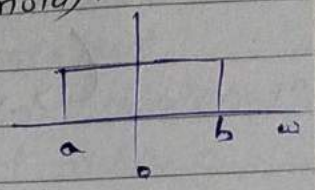


fan-in = 3
fan-out = 2

idea

② uniform init :- (works fairly well for sigmoid) pdf

$$w_{ij}^{k} \sim \text{Uniform}\left(\underbrace{\frac{-1}{\sqrt{fan\text{-}in}}}_{a}, \underbrace{\frac{1}{\sqrt{fan\text{-}in}}}_{b}\right) \Rightarrow$$



NOTE :- No concrete agreement with in all researchers. Every researcher comes with their own init method and it works fairly well.

idea ③ Xavier/Glorot init

Normal Xavier/Glorot a) $w_{ij}^{k} \sim N(0, \sigma)$ $\sigma_i = \frac{2}{fan\text{-}in + fan\text{-}out}$

Uniform Xavier/Glorot b) $w_{ij}^{k} \sim U\left(\frac{-\sqrt{6}}{\sqrt{fan\text{-}in + fan\text{-}out}}, \frac{\sqrt{6}}{\sqrt{fan\text{-}in + fan\text{-}out}}\right)$

idea ④ He-init (2015) (works well for ReLU and Leaky ReLU)

normal → a) $w_{ij}^{k} \sim N(0, \sigma)$ $\sigma = \sqrt{\frac{2}{fan\text{-}in}}$

uniform → b) $w_{ij}^{k} \sim U\left(-\sqrt{\frac{6}{fan\text{-}in}}, +\sqrt{\frac{6}{fan\text{-}in}}\right)$

⊛ Batch normalization :- (2015)

$x \rightarrow L_1 \rightarrow L_2 \rightarrow L_3 \rightarrow L_4 \rightarrow L_5 \rightarrow$ Output layer $\rightarrow y$

→ this problem is known as internal covariance shift

x : mini batch (mini batch SGD) also data is normalized

b_1 : for layer 1 data is properly normalized & distribution of batch b_1 and b_2 will be same

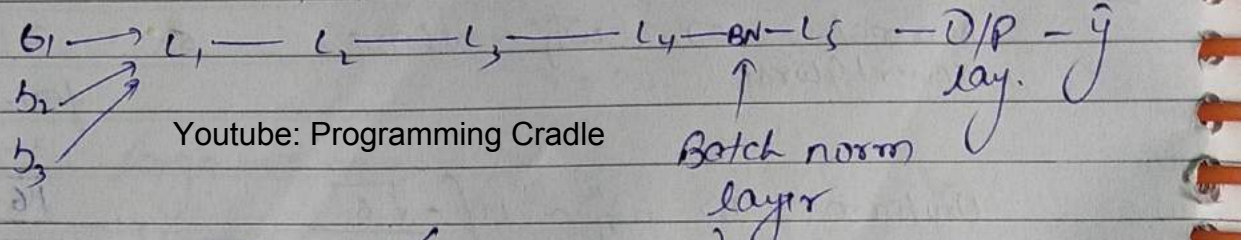
b_2

for layer 5 data in diff batches will may have diff dist. bcoz normalizat" is not done at each layer.

Solution for internal cov. shift :-

↳ -> We add addition layer known as batch normalization layer. In this layer only current batch is normalized based on batch's mean and std-dev.

-> Usually batch layer is used deep inside the network.



let $b_1 = \{x_1, x_2 \ldots x_n\}$ — norm will be done for only these pts. not on whole data.

NOTE :- BN can be added btw any layers.

All are diff^n func. backprop can be applied to learn γ, β

Input :- Values of x over a mini-batch : $\mathcal{B} = \{x_1 \ldots x_m\}$
parameters to be learned γ, β

Output :- $\{\tilde{x}_i : BN_{\gamma,\beta}(x_i)\}$

$\mathcal{B} \rightarrow x_i \rightarrow [BN] \rightarrow \tilde{x}_i \; \tilde{\mathcal{B}}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \quad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \quad \text{// mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad \text{// normaliz}^n$$

ϵ small value to avoid divid. by zero

$$\tilde{x}_i \leftarrow \underbrace{\gamma x_i}_{\text{scale}} + \underbrace{\beta}_{\text{shift}} \equiv BN_{\gamma,\beta}(x_i) \quad \text{// scale and shift}$$

-> Advantages of BN :-

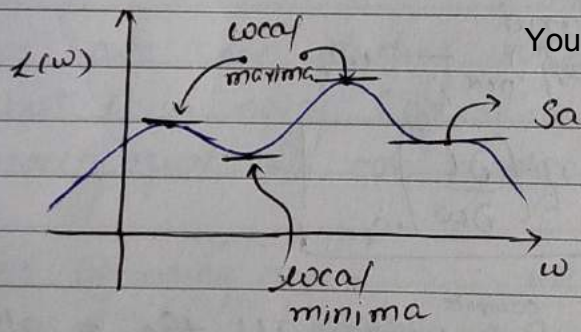
-> faster convergence because it is guaranteed that all the batches will have same distribut^n and because of it we can afford to have larger learning rate. (η)

$$(\omega_{ij}^{l})_{new} = (\omega_{ij}^{l})_{old} - \eta \left(\frac{\partial L}{\partial \omega_{ij}^{l}}\right)$$

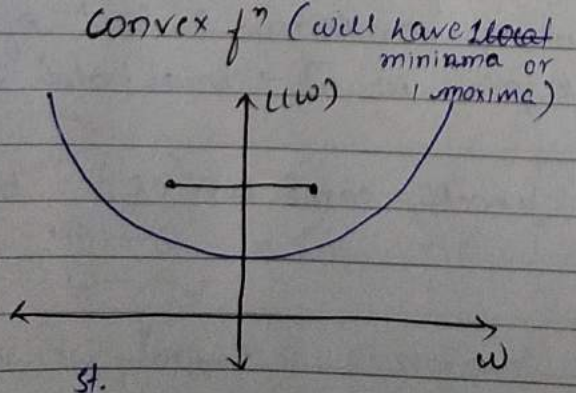
-> BN also works as weak ~~batch~~ regularization so we should use BN in addition to dropouts.

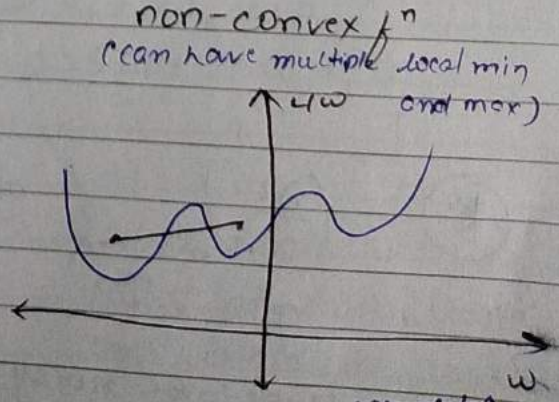
-> Avoids internal cov. shift. and hence deep NN can be trained.

⊛ Optimizers :- Hill descent analogy in 2D



NOTE :- SGD or mini-batch SGD could get stuck at saddle point.





if a line connecting 2 points in the same region lies completely in one region then it is convex f^n else non-convex.

NOTE :- → for Logistic Regression, Linear Regression and SVM loss f^n can be shown as convex f^n and hence SGD or batch SGD works fine.

→ Where as in MLP loss f^n is non convex and can have multiple local minima and local maxima. Also it can have saddle point and hence SGD or both SGD won't always work.

⊛ SGD and momentum

in each iter. $(w_{ij}^k)_{new} = (w_{ij}^k)_{old} - \eta \left[\frac{\partial L}{\partial w_{ij}^k}\right]_{(w_{ij}^k)_{old}}$ } update f^n

for simplicity $(w_{ij}^k)_{new} \rightarrow w_t$

and

$(w_{ij}^k)_{old} \rightarrow w_{t-1}$

$$\boxed{w_t = w_{t-1} - \eta \left[\frac{\partial L}{\partial w}\right]_{w_{t-1}}}$$

$\Omega = \{x_i, y_i\}_{i=1}^n$ $\frac{\partial L}{\partial w}$ $\xrightarrow{\text{compute}}$ using all the n pts in Ω, it is called as GD

→ compute using one pt chosen at random, known as SGD

(⊛ mostly used) → using random subset of k pts in Ω → mini batch SGD

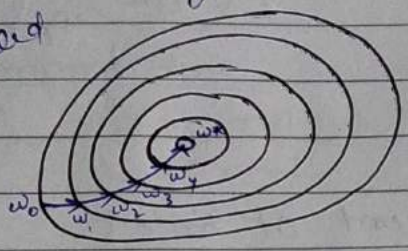
⊛ $\left(\frac{\partial L}{\partial w}\right)_{\text{mini-SGD or SGD}}$ are not exactly same as $\left(\frac{\partial L}{\partial w}\right)_{GD}$ but close estimates

↑ tends to be noisy.

ideally we want

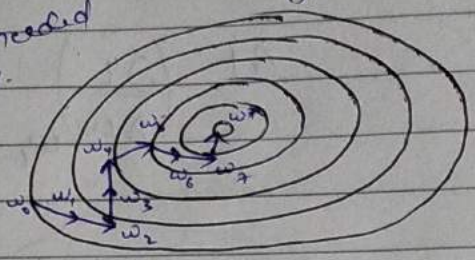
GD convergence

5 iter needed to conv.



SGD convergence

8 iter needed to conv.



NOTE :- To get the "almost" same w^* as in GD, SGD requires more no. of iterations.

w^*_{GD} ——requires——> 5 iter.

similar → SS

w^*_{SGD} ——requires——> 20 iter may be, or even more

Batch SGD with momentum :-

↳ exponential weighting

As we have seen finding w^* using SGD work well but it is noisy. (as w^*_{GD} and w^*_{SGD} are approx. equal not exactly equal) We can apply denoise methods.

Denoise by using exp. wt. avgs :-

date pts. → a_1 a_2 a_3 .... t_1 t_2 t_3 ....

$t=1$ $V_1 = a_1$

$t=2$ $V_2 = \gamma V_1 + a_2$

$t=3$ $V_3 = \gamma V_2 + a_3 = \gamma(\gamma V_1 + a_2) + a_3$

$= \gamma^2 a_1 + \gamma a_2 + a_3$

let $\gamma = 0.5$

$= 0.25 a_1 + 0.5 a_2 + 1 a_3$

t_1 t_2 t_3

$\therefore$ Giving more wt. to latest pt. and lesser and lesser to previous pts.

$0 < \gamma < 1$ $\Rightarrow$

$$\begin{cases} V_1 = a_1 \\ V_t = \gamma V_{t-1} + a_t \end{cases}$$

$V_t \rightarrow \approx$ denoised estimate.

Youtube: Programming Cradle

mini-batch SGD → $\omega_t = \omega_{t-1} - \eta \left(\frac{dL}{d\omega}\right)_{\omega_{t-1}}$

let $\left(\frac{\partial L}{\partial \omega}\right)_{\omega_{t-1}} = g_t$

↑ gradient at time t

Stol. update eqn ⇒ $\omega_t = \omega_{t-1} - \eta g_t$ ① → mini batch SGD

eqn ① modified using expon. weighted. $0 < \gamma \leq 1$

$$v_t = \gamma v_{t-1} + \eta g_t$$
$$\omega_t = \omega_{t-1} - v_t$$

good value of $\gamma = 0.9$ (thumb rule)

case 1 $\gamma = 0$; $v_t = \eta g_t$ ⇒ $\omega_t = \omega_{t-1} - \eta g_t$

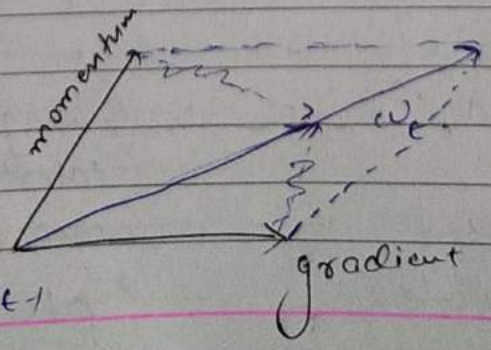
case 2 $\gamma = 0.9$; $v_t = 0.9 v_{t-1} + \eta g_t$; $\omega_t = \omega_{t-1} - (0.9 v_{t-1} + \eta g_t)$

⊛ SGD + momentum = exp. weighting to denoise your SGD gradients.

$$\omega_t = \omega_{t-1} - [\gamma v_{t-1} + \eta g_t]$$

↑ momentum ↑ gradient

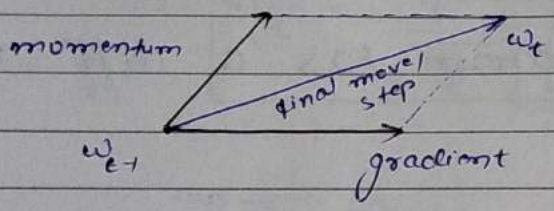
Speeds up convergence



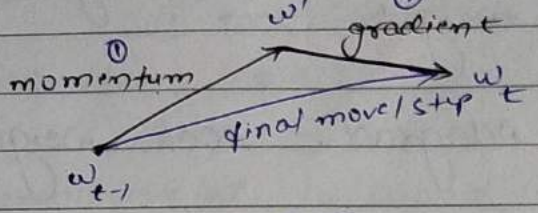
⊛ Nesterov Accelerated Gradient (NAG)

⊛ SGD + momentum

$$\omega_t = \omega_{t-1} - (\underbrace{\gamma V_{t-1} + \eta g_t}_{V_t})$$



⊛ NAG



$$\Rightarrow \omega_t = \omega_{t-1} - (\gamma V_{t-1} + \eta g')$$

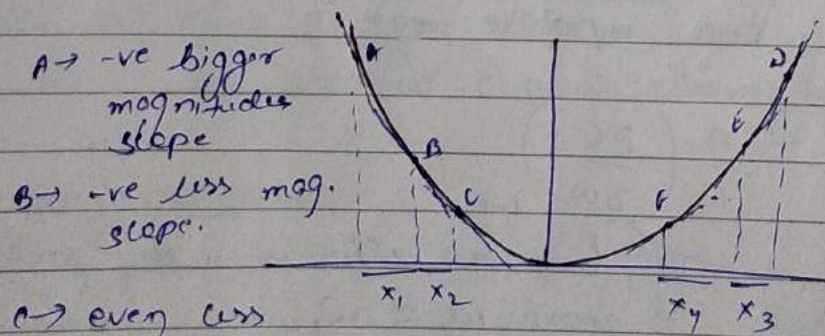
$$\Rightarrow g' \neq g_t$$

$$\Rightarrow g' = \left(\frac{\partial L}{\partial \omega}\right)_{\underbrace{\omega_{t-1} - \gamma V_{t-1}}_{\omega'}}$$

Q) Why gradient and momentum has diff. directions?

Ans) → Momentum gives direction based on previous gradient / slope, so if there are many +ve slopes steps in -ve direction will be bigger and if no. of -ve slopes are more, steps in +ve dirn will be bigger.

→ Gradient gives direction based on magnitude and polarity of the slope. If slope is large -ve value, step in +ve direction will be bigger and if slope is small -ve value, step in +ve direction will be smaller.



$slope_D > slope_E > slope_F$

∴ As previously there are more +ve slopes (D, E) larger step has been taken $x_4 > x_3$

∴ As we can see $x_4 > x_2$

Adagrad (Adaptive gradients)

* For SGD and SGD + momentum learning rate (η) is constant (very small no. 0.01) for each weight.

* Adagrad :- each weight / param has a diff^n η

Q) Why have diff^n η?
A) Because there can be features which are sparse and dense, so they need diff^n learning rates.

eg. in linear regression (loss = sqrd loss)

$$\left(\frac{\partial L}{\partial w}\right)_{w_{t-1}} = \sum_{i=1}^{n} (-2x_i)(y_i - w_{t-1} x_i)$$

sparse — dense

most of times will be '0' and hence $\frac{\partial L}{\partial w}$ will be very small	most of times values will be non zero hence $\frac{\partial L}{\partial w}$ will be large.

Now lets see our update eq^n

$$w_t = w_{t-1} - \eta \left(\frac{\partial L}{\partial w}\right)_{t-1}$$

$w_t \neq w_{t-1}$ for large features ← → as this is very small for sparse, $w_t \approx w_{t-1}$

Youtube: Programming Cradle

SGD : $w_t = w_{t-1} - \eta g_t$ → same for all wts.

Adagrad : $w_t = w_{t-1} - \eta'_t g_t$ → diff^n η' for each feature @ each iteration.

$$\Rightarrow \boxed{\eta'_t = \frac{\eta}{\sqrt{\alpha_t + \epsilon}}}$$

η → constant (0.01)

ϵ → small +ve no. to avoid divid. by 0

$$\alpha_t = \sum_{i=1}^{t-1} g_i^2 \quad (\alpha_t \uparrow \text{ +ve}, \; g_i^2 \hookrightarrow \text{+ve})$$

$$g_i = \left(\frac{\partial L}{\partial w}\right)_{w_{i-1}}$$

and $\alpha_t \geq \alpha_{t-1} \Rightarrow t \uparrow \Rightarrow \alpha \uparrow \Rightarrow \eta'_t \downarrow$ (t ↑ = iteration)

⊛ Advantages :-
- no need of manually tune η
- chooses right value of η based on previous grad for sparse and dense features

⊛ Disadvantages :-
- α_t can become very large as $t \uparrow$ and because of this convergence will be slower.

⊛ Adadelta and RMS Prop.
- These 2 algos help in getting rid of disadvantage of adagrad (slow convergence)

⇒ Ada delta :-

$$w_t = w_{t-1} - \eta'_t g_t$$

$$\eta'_t = \frac{\eta}{\sqrt{eda_t + \epsilon}} \; ; \quad eda_t = \gamma * eda_{t-1} + (1-\gamma) g_t^2$$

→ exponential decay avg.

Adadelta: exp. wt. avg. of g_i^2 instead of sum of g_i^2 (adagrad) so as to avoid large denominator in η_t hence slow conv. can be avoided.

⊛ Adam:- (Adaptive Moment Estimation)

eda g_t : $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$ ① (0.9) $0 \leq \beta_1 \leq 1$

eda g_t^2 : $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$ ② $0 \leq \beta_2 \leq 1$

$$\hat{m}_t = \frac{m_t}{1-(\beta_1)^t} \quad ; \quad \hat{v}_t = \frac{v_t}{1-(\beta_2)^t}$$ } bias correction

$$w_t = w_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

mean → 1st order moment
var → 2nd order moment

typically $\alpha = 0.001$

Q> which optimizer / algorithem to choose ?

Mini Batch SGD → Small / shallow NN

↓

Momentum based NAG → works well in most cases but slower

↓

Adagrad → ⊛ sparse ⊛

↓

Adadelta & RMS prop → sparse

↓

Adam → Best of all mentioned above

⊛ Gradient Monitoring and clipping:-

↳ It is a good habit to monitor gradients and updates for each epoch and weights and layers. it will help in knowing if vanishing grad. prob. is occuj or not., exploding prob or any other prob.

⊛ To get rid of exploding gradient we can use clipping

$$Grad_{new} = \frac{Grad}{||Grad||_2} * \tau$$

τ ↑ threshold.

$$||Grad||_2 = \sqrt{g_1^2+g_2^2+g_3^2+g_4^2+g_5^2}$$

Let Grad = | 2 | 5 | 10 | 1000 | 63 |
(g_1 g_2 g_3 g_4 g_5)

$\frac{Grad}{||Grad||_2}$ = | | | | | |

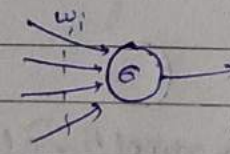
< 1

⇒ $Grad_{new}$ will always be less than τ (threshold)

⊛ Softmax and Cross-entropy for multi-class classifⁿ

Softmax classifier = logistic regⁿ + multi-class

⇒ LR



$P(y_i=1|x_i) = \hat{y}_i = \sigma(z) = \frac{1}{1+e^{-z}}$

Softmax classifier

$D : \{x_i, y_i\}$ multi-class

$y_i \in \{1, 2, 3 \ldots K\}$

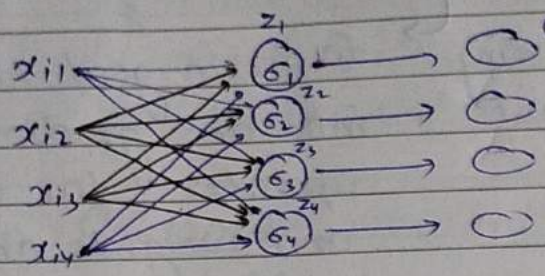
$x_i \rightarrow$ [M] → $P(y_i=1|x_i)$
$P(y_i=2|x_i)$
$P(y_i=3|x_i)$
⋮
$P(y_i=K|x_i)$

M ← model

Sum → 1

⊛ Cross-entropy :- it can be used to define a loss f^n in ML and optimizat^n, it is defined on prob. dists, not single values.

⊛ Softmax at K = 4



$$z_1 = \sum_{j=1}^{d} x_{ij} w_{j1} \; ; \; z_2 = \sum_{j=1}^{d} x_{ij} w_{j2} \; \ldots$$

$$\sigma_1(z_1) = \frac{e^{z_1}}{\sum_{i=1}^{K} e^{z_i}} \; ; \; \sigma_2(z_2) = \frac{e^{z_2}}{\sum_{i=1}^{K} e^{z_i}} \; \ldots$$

$$\Rightarrow \sigma_1(z_1) + \sigma_2(z_2) + \ldots + \sigma_K(z_K) = 1$$

$$\Rightarrow \frac{e^{z_1}}{\sum_{i=1}^{K} e^{z_i}} + \frac{e^{z_2}}{\sum_{i=1}^{K} e^{z_i}} + \ldots \frac{e^{z_K}}{\sum_{i=1}^{K} e^{z_i}}$$

$$\Rightarrow \frac{e^{z_1} + e^{z_2} + \ldots e^{z_K}}{\sum_{i=1}^{K} e^{z_i}}$$

$$\Rightarrow \frac{\sum_{i=1}^{K} e^{z_i}}{\sum_{i=1}^{K} e^{z_i}} = 1$$

⊛ Softmax → generalization of LR to multi-class setting.

LR :- $\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{e^z+1}$

↓ minimizes log-loss

Softmax :- $\sigma_1(z_1) = \frac{e^{z_1}}{\sum_{i=1}^{K} e^{z_i}}$; $\sigma_1, \sigma_2 \ldots \sigma_K$

↓ minimizes multi-class log loss

(*) Multi class log-loss

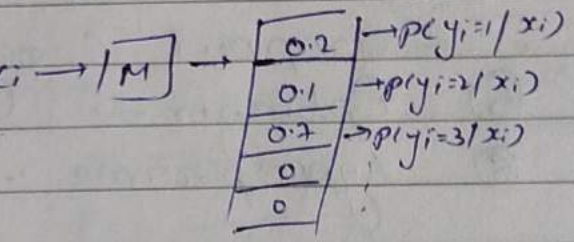
$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{K} y_{ij}\log p_{ij}$$

(*) 2 class log-loss

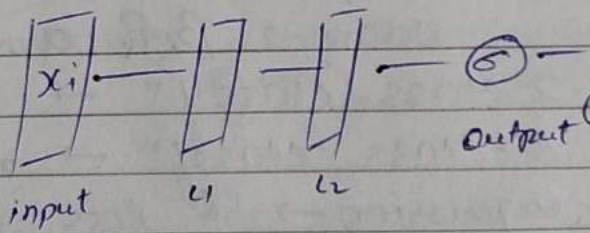
$$-\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log p_i + (1-y_i)\log(1-p_i)\right]$$

x_i, $y_i = 3$

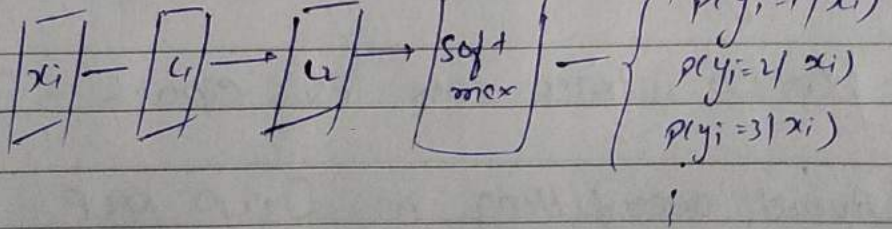
1	2	3	...	K
0	0	1	...	0
y_{i1}	y_{i2}	y_{i3}		y_{iK}



=> 2 class classfn



-> K class classfn



(*) In case of regression we can use linear unit instead of sigmoid

$f(z) = z$

↑

linear unit

(*) "How to" train an MLP?

1. Preprocess :- Data Normalization
2. Weight initialization :- ✓ Xavier / Glorot → Sigmoid / tanh
   :- ✓ He → ReLU
   :- Gaussian (small value of σ)
3. Choose activation f^n :- ReLU
4. Add Batch normalization especially for deep MLP (later layers)
   dropout (regularization)
5. Optimizer → ADAM
6. hyperparams :- Architecture → no. of layers
   → no. of neurons
   :- Dropout rate
   :- Adam → β_1, β_2, α
7. Loss f^n → 2 class classif^n → log loss
   → K-class classif^n → multi class log loss
   → regression → SQ. loss
8. (*) Always monitor gradients and apply clipping.
9. plots : train loss vs epoch
10. Avoid overfitting in Deep MLP

NOTE :- Its good to have fewer neurons in later layers as compared to earlier layers.

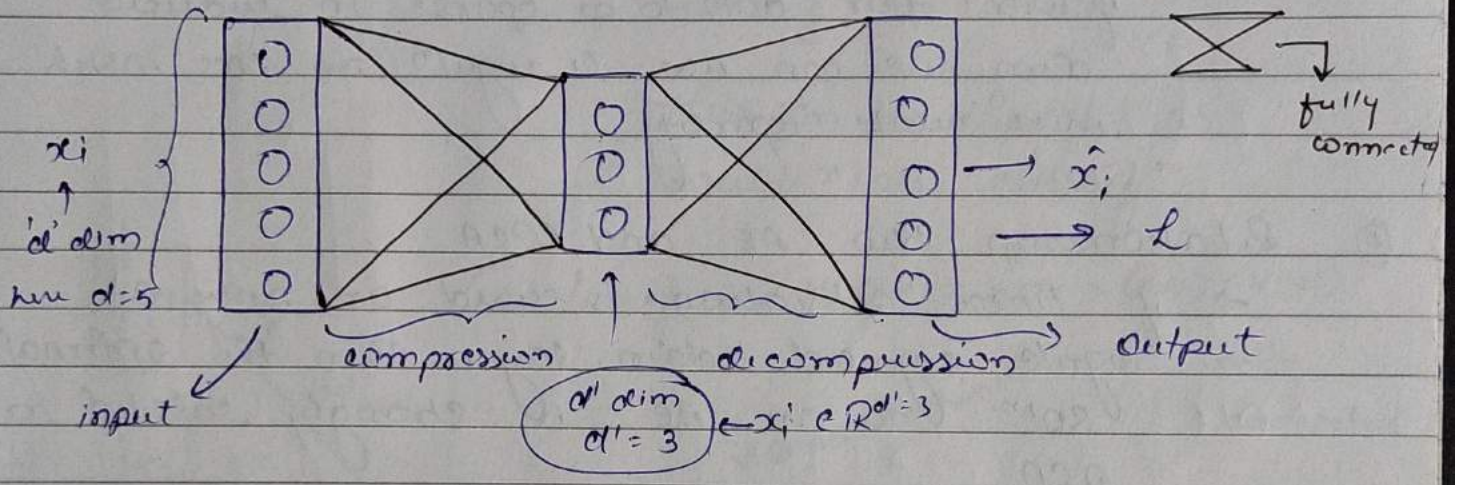
⊛ Type of NN :- Auto encoder

→ Auto encoder is a NN which performs dim. reduction

→ objective :- $D = \{x_i\}_{i=1}^{n}$ $x_i \in R^d$

↓ $d' < d$

$D' = \{x_i'\}_{i=1}^{n}$ $x_i' \in R^{d'}$



⊛ If $\hat{x}_i \approx x_i$ we can say that d' (3) is sufficient enough to represent data in d (5) dim

$\mathcal{L}(x_i, \hat{x}_i) = \|x_i - \hat{x}_i\|^2$ ideally $\mathcal{L} = 0 \Rightarrow x_i = \hat{x}_i$

↑ loss fⁿ

NOTE :- We can have multiple hidden layers instead of just 1.

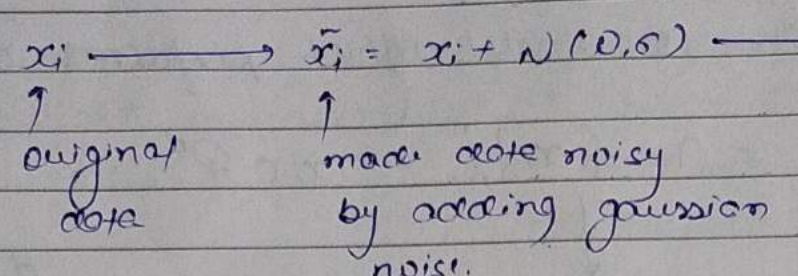
⊛ Denoising Auto Encoder :-

$D = \{x_1, x_2, x_3 \ldots x_n\}$ ← actual data

We get → $\tilde{D} = \{\tilde{x}_1, \tilde{x}_2, \tilde{x}_3 \ldots \tilde{x}_n\}$

let $\tilde{D}$ is noisy

Steps in denoising AE :-



* Sparse Encoders :-
  if want our data to be sparse in reduced dim we can use L1 regularn on loss which will create sparsity.

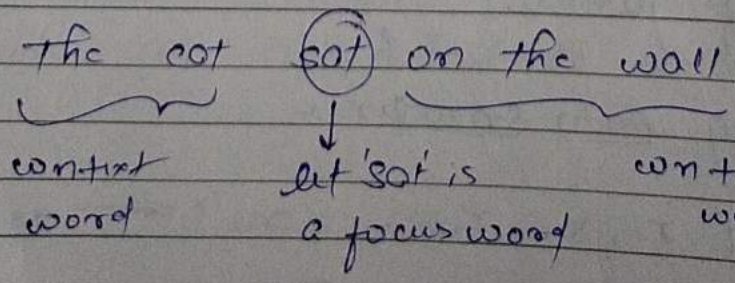
* Relationship b/w AE and PCA
  → if linear activation is used or only a single sigmoid hidden layer, then the optimal soln to an AE is strongly related to PCA

* Word2Vec CBOW (continuous bag of words)

  focus word :- word for which vec is being generated.

  context word :- All the other words in the sentence

  eg.

① ⇒ $V \rightarrow$ dictionary / Vocabulary of words
⇒ $v \rightarrow \dim(V)$

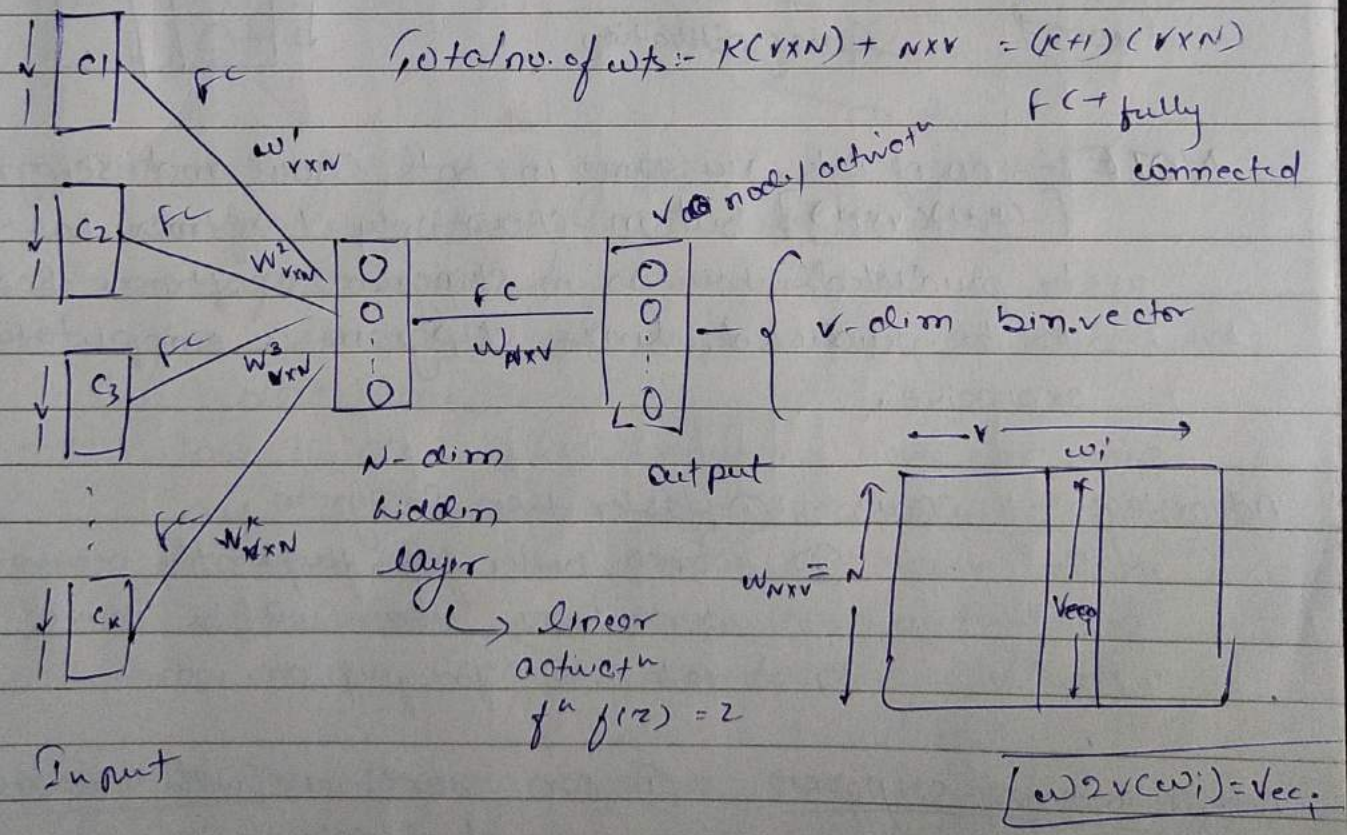
② One hot encoding each word
$w_i \in R^v$: binary vector of V-dim.

⇒ Core Idea of CBOW :- Given Context words, can we predict focus word

↓

This can be thought as multi class classif^n

Youtube: Programming Cradle



Total no. of wts :- $K(V\times N) + N\times V = (K+1)(V\times N)$

FC → fully connected

↳ linear activat^n $f^n f(z) = z$

$w2v(w_i) = Vec_{w_i}$

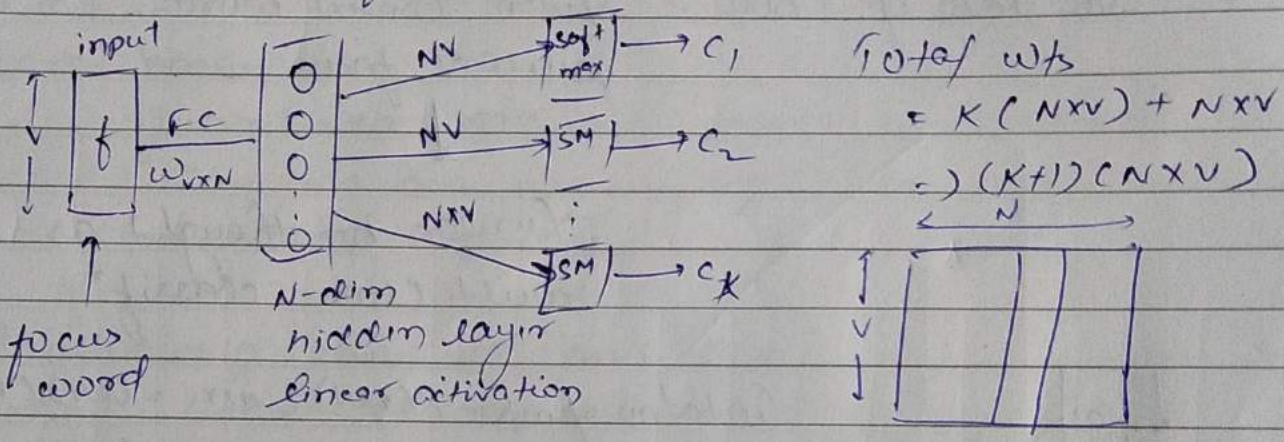
dataset will look something like this :

focus word	context word	Sent : $w_2\ w_1\ w_4\ w_3\ w_5$
w_1	$w_2\ w_3\ w_4\ w_5$	
w_2	$w_1\ w_3\ w_5\ w_4$	

⊛ Word2Vec : Skipgram

CBOW :- predict focus word given context words

Skipgram :- predict context words given focus word.



NOTE :- no. of wts are same in both CBOW and Skipgram [(K+1)(VxN)] but in CBOW only 1 softmax has to be predicted whereas in Skipgram K softmaxis has to be predicted. Hence skipgram is computationally expensive.

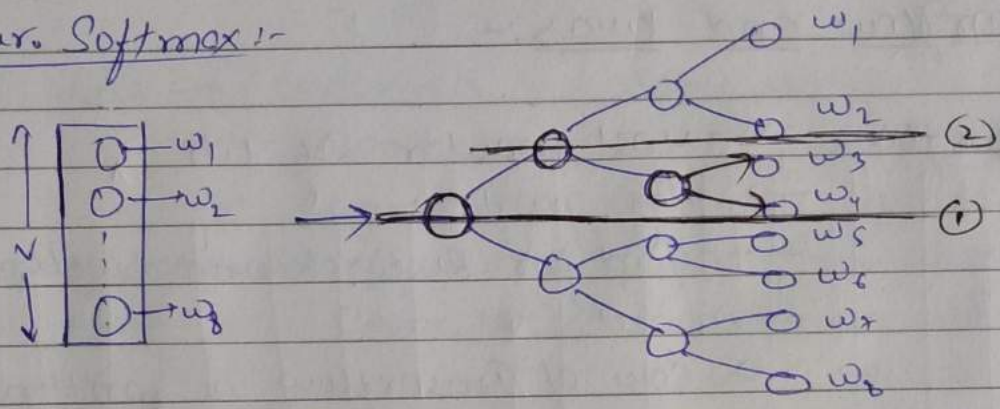
Advantages of CBOW :- ① faster than Skipgram
② works better for frequently occurring words bcoz there will be more data points for freq. occ. words.

Advantages of skipgram :- ① can work well with smaller amount of data
② works well for infrequent words.

* Major problem with CBOW and skipgram is they have so many wts. $(V+1)(V \times N)$

* Algorithmic Optimizations:-
  - ↳ hierarchical Softmax
  - ↳ Negative Sampling.

* Hier. Softmax:-



→ instead of linearly going through all the V-class/wts we can use binary tree data structure

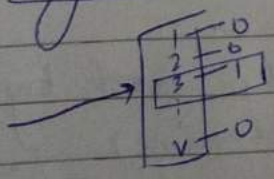
→ at first node if $p(y_i = w_3 / x_i)$ is more then we can skip all the nodes below block line ①.

→ Now if $p(y_i = w_3 / x_i)$ is more we can skip all the nodes above block line ②

→ at last node we can select our class.

* Here we are computing on $\log_2(V)$ instead of V wts.

* Negative Sampling:- update only a sample of words pr iteration



sampling:- ① always keep the target word. ($word_3$)

② amongst all the non target words update (wt. of) don't all, update sample from & repeat.

Sample is selected based on below prob.

$$p(w_i) = 1 - \sqrt{\frac{\tau}{freq.(w_i)}}$$

τ → threshold typically 10^{-5}

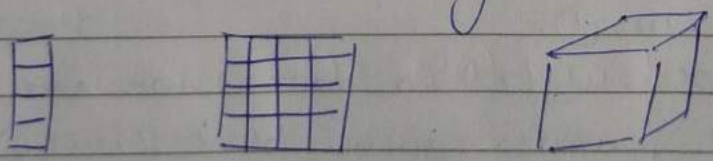
Probability of a word getting selected in the sample will be more if it occurs more frequently.

Youtube: Programming Cradle

⊛ Tensorflow and Keras :-

Tensorflow → Most popular DL lib
→ By google
→ Good for Research and development as well.
→ Core of Tensorflow is written in c/c++
→ Has interface with Py., Java, JS
→ Tensorflow lite → can run on android
→ learning curve is steep.

⊛ Tensor :- n-dim array.



1D array/tensor 2D array/tensor 3D-tensor

⊛ Flow → Tensor's are flowing from i/p layer to o/p layer.

⊛ Keras :- → Simple
→ High-level NN-lib (mostly used by developers)
→ Easy to learn
→ fewer lines of code.

-> learning.tensorflow.com
-> cloud.google.com learn duplex without phd.

-> usr → keras → tensorflow → exe
keras ↑ front end
tensorflow ↑ back end
keras → theano → exe

* Other alternatives of TF :- theano, caffe2, pytorch, MXNET

* keras2 is deeply integrated with TF
↳ Same low level control.

* TF has → constants } Same as in
→ variable } Python
→ placeholders. => stores mini-batches, once the processing is done on one batch, next batch will be stored in placeholder

=> x = tf.placeholder(tf.float32, [None, 784])

x = [784] → stores value of ~~size~~ type float.

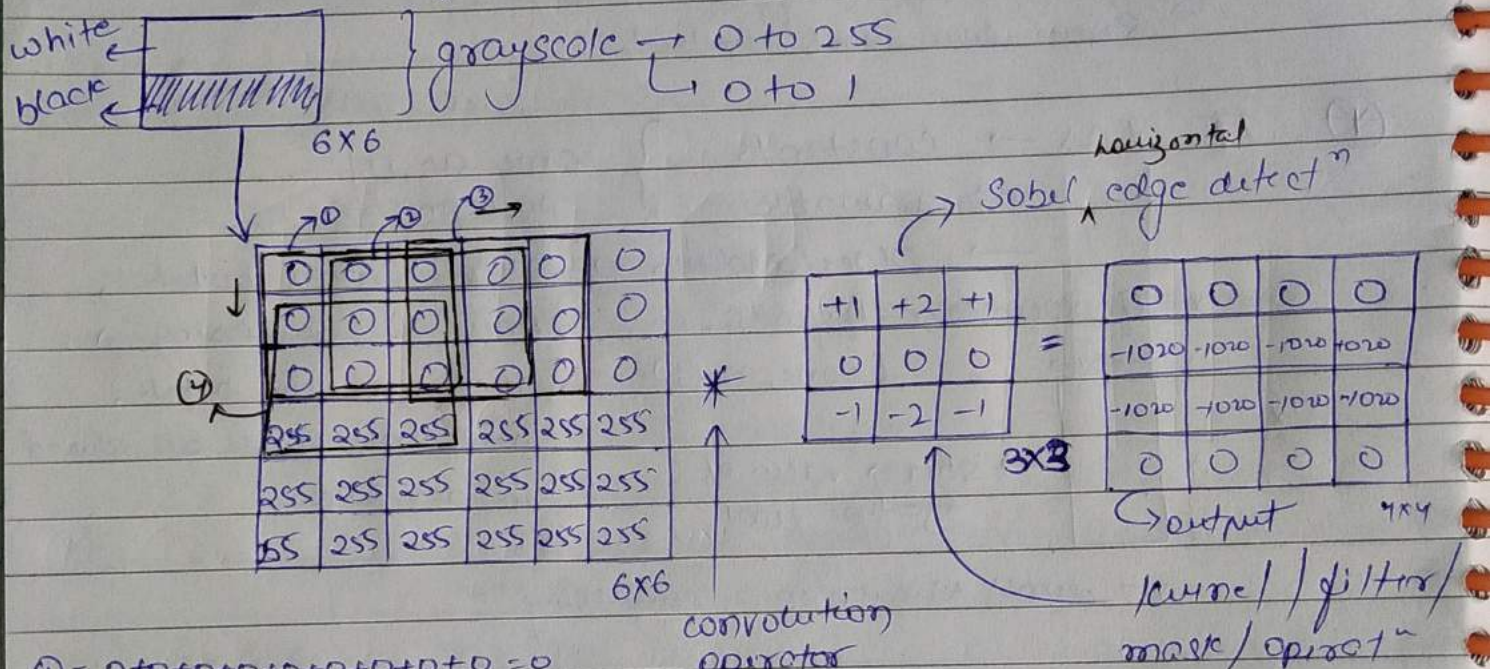
y → don't know, can be any thing

=> w = tf.Variable(tf.zeros([784, 10]))
=> b = tf.Variable(tf.zeros([10]))

⊛ Convolutional Neural Nets (CNN, convNets)

↳ used for visual tasks (MNIST)
(computer vision)

⊛ Convolution Edge Detection on Images:



① = 0+0+0+0+0+0+0+0+0 = 0

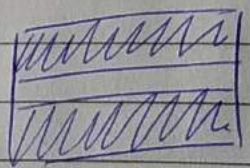
② = 0 ③ = 0

④ = 0+0+0+0+0+0+(-255)+(-510)+(-255) = -1020

output —normal^n→

255	255	255	255
0	0	0	0
0	0	0	0
255	255	255	255

∴



⊛ Sobal vertical edge det^n

4X4

+1	0	-1
+2	0	-2
+1	0	-1

input (n×n) $\xrightarrow{(k\times k)\ \text{kernel}}$ (n-k+1)(n-k+1) output

eg:- 6×6 * 3×3 = 4×4 (reduced size img.)

✱ Convolution Padding:

→ If we don't want to reduce the size of the output image we can apply padding on the input image.

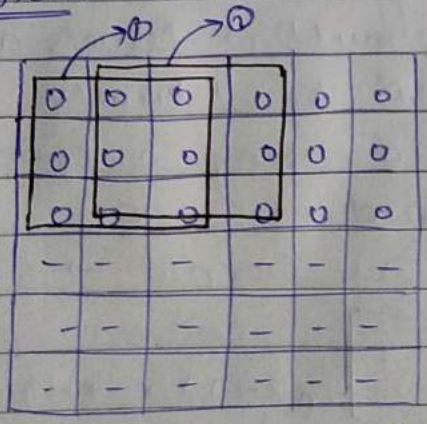
① Zero-padding (most often used)

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	255	255	255	255	255	255	0
0	255	255	255	255	255	255	0
0	255	255	255	255	255	255	0
0	0	0	0	0	0	0	0

② Same value padding
↳ pad with same value as nearest value.
(this tech. is not used much)

$\Rightarrow n\times n \xrightarrow[\text{with padding } p]{k\times k} (n-k+2p+1)(n-k+2p+1)$

✱ Strides :-



– 255

from ① to ② Stride of 1 pixel
it can be stride of 2 or more pixels as well.

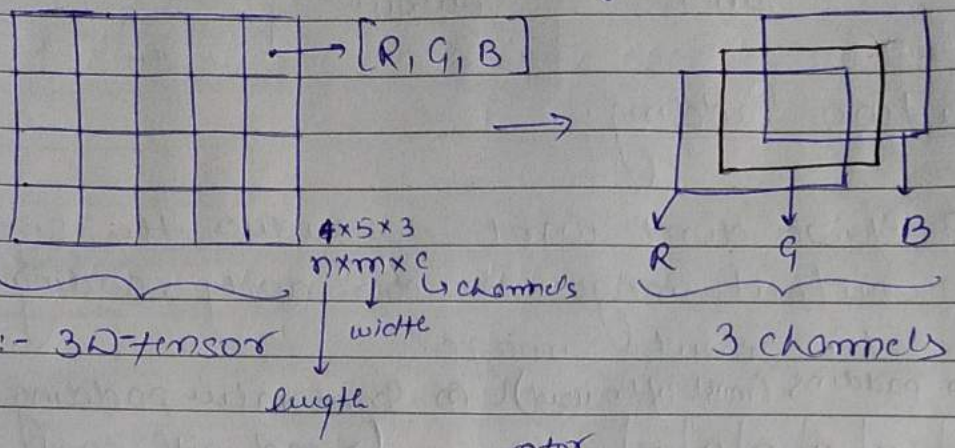
$n\times n \xrightarrow[k\times k]{\text{Stride}=s} \left(\left\lfloor\frac{n-k}{s}\right\rfloor+1\right)\times\left(\left\lfloor\frac{n-k}{s}\right\rfloor+1\right)$

n×n
n=6

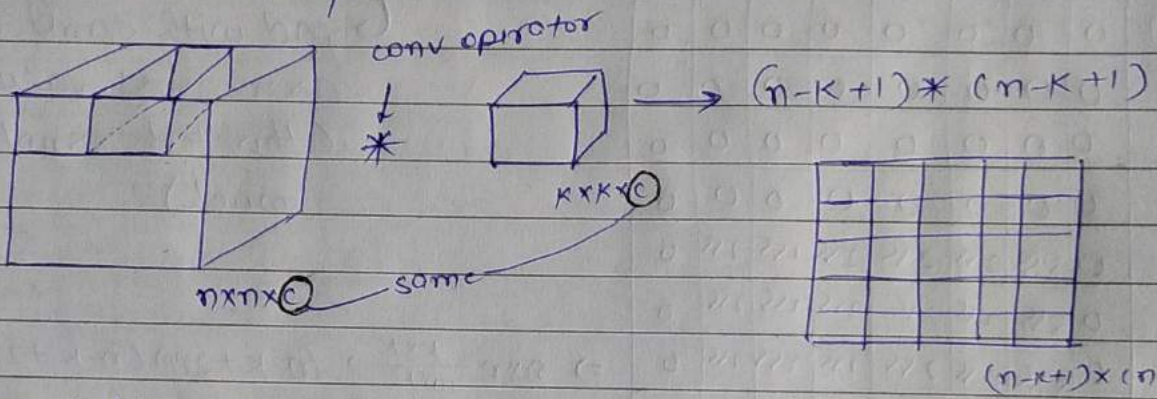
eg:-

$6\times6 \xrightarrow[s=2]{k=3} 2\times2$

✱ Convolution on color-images :-



Color img :- 3D-tensor



✱ Convolution layer in NN

① Conv. layer are inspired by biological brain

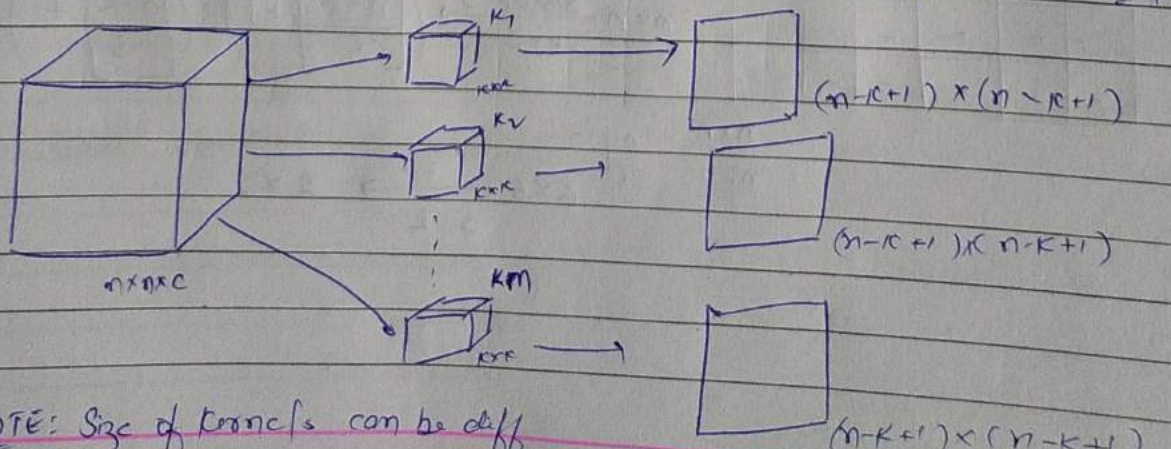
② for multiple edge detection → multiple kernels are needed

1 edge detected by 1 kernels.

in CNN Kernels are learned using backpropagation

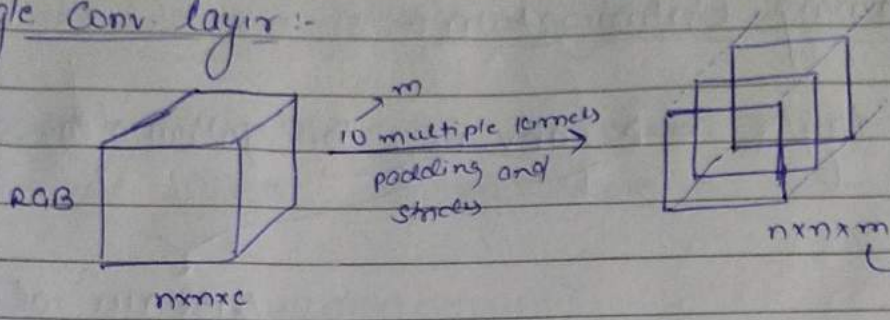
in MLP :- w's are learned → $w.x$

in CNN :- kernels matrices are learned → $\Omega * \omega_{K\times K}$

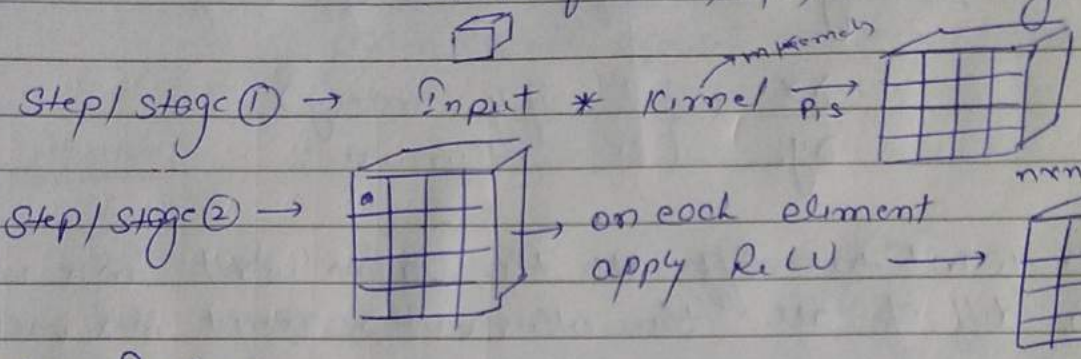


NOTE: Size of kernels can be diff

Single Conv layer :-



m (no. of kernels); p ; s are hyperparam.



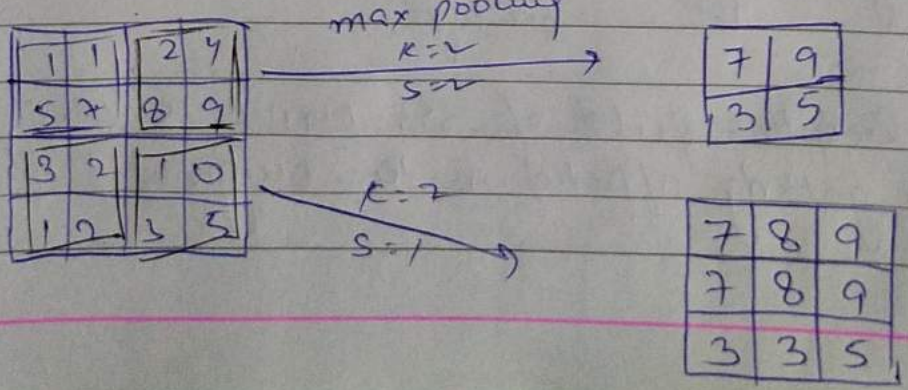
⊛ Max-Pooling

-> location invariance :- where the object is in the image, it should be detected

-> Scale invariance :- what ever is the size of the obj. it should be detected.

-> rotation invariance :- Even if the obj is rotated, it should be detected.

=> Max-pooling helps in making the model invariant to above 3 invariance. (locat^n, rotat^n, scale)



⊛ CNN Training Optimization :-

→ Conv-layer ⟹ Conv operation followed by ReLU

both are differentiable

→ Max-pooling ⟹ differentiation of max f^n

$$\frac{\partial x}{\partial y_i} = \begin{cases} 1 & \text{if } y_i = \max(y) \\ 0 & \text{otherwise.} \end{cases}$$

⇒ We want to diff all the values w.r.t max value so, diff of all the other values except max value will be zero and diff of max value wrt max value will be One.

Let $\begin{vmatrix} a & b \\ c & d \end{vmatrix}$ Let b is max value

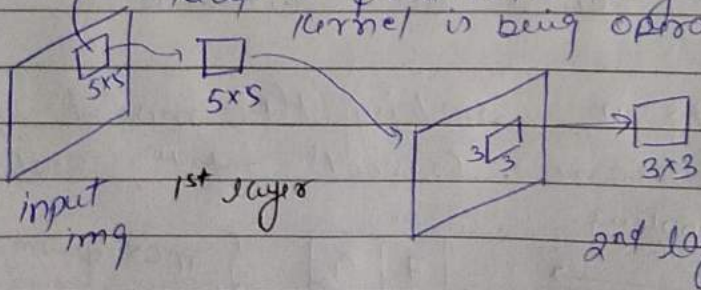
$$\frac{\partial a}{\partial b} = 0, \quad \frac{\partial b}{\partial b} = 1, \quad \frac{\partial c}{\partial b} = 0, \quad \frac{\partial d}{\partial b} = 0$$

Youtube: Programming Cradle

⊛ Receptive field and effective Receptive field

multiple layer

→ Receptive field is the region/part of img on which kernel is being operated at a given point of time



effective Receptive field of 3×3 kernel is the region getting affected in the original I/P img.

Data Augmentation :-

adding several diff variations of original data like flipping, rotation, zoom in, zoom out version of the data

Why data augmentation?

→ invariance (any 3 invariance)

→ from small dataset to large dataset.

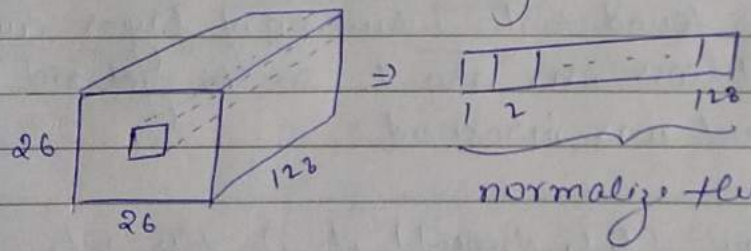
AlexNet

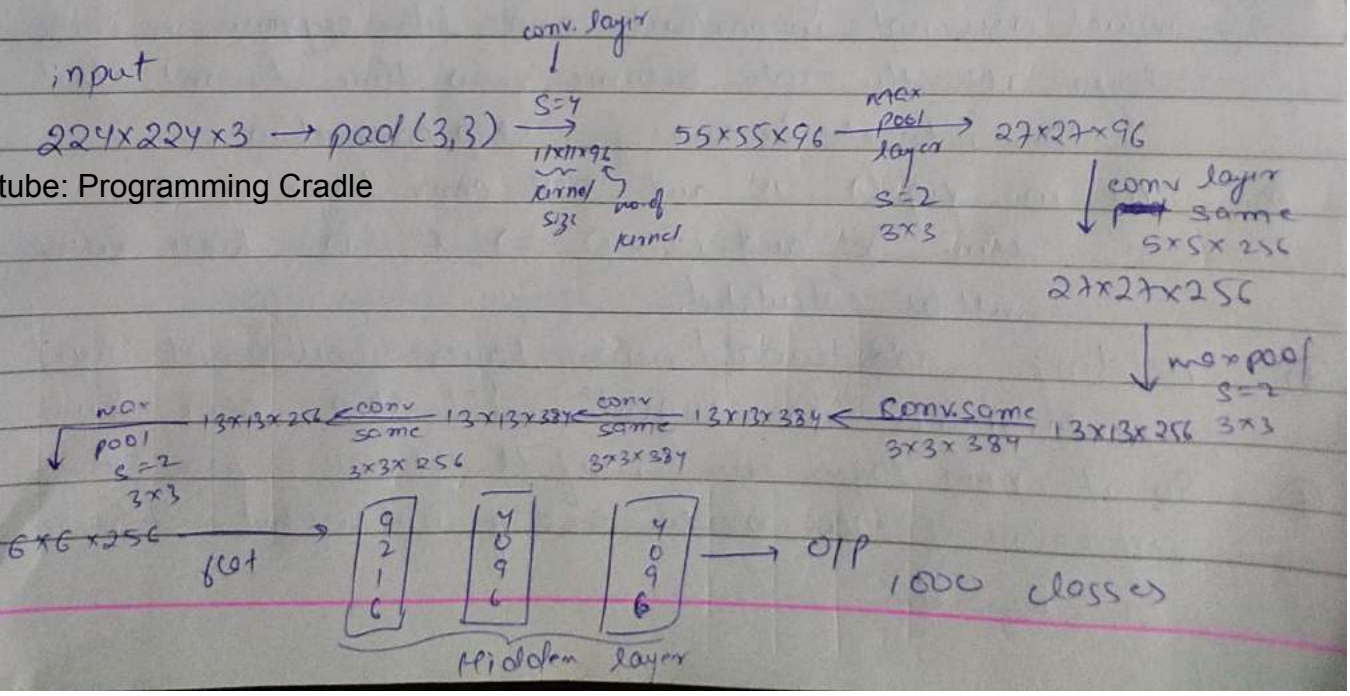
↳ used ReLU as activation f^n

↳ dropouts were used

↳ GPUs were used

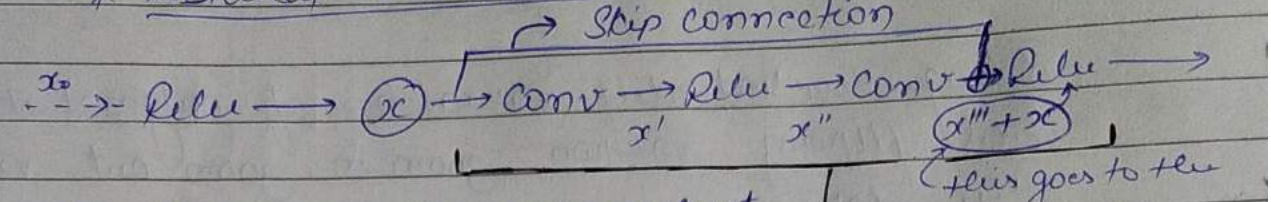
→ local response normalizn (lrn) (but not used much, batch norm. is the way to go)





* Residual Network (ResNet) *

fig ①



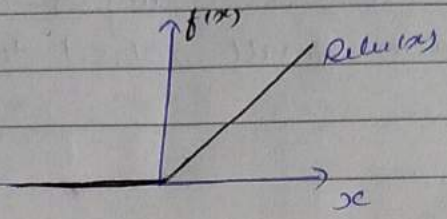
→ conv layer can be dot product (mcp) as well.

identity block when x is i/p

} at it returns x if the conv layers are not useful

* Size of x''' and x should be same (use padding)

→ Relu fⁿ graph



Youtube: Programming Cradle

① { x: +ve → Relu(x) = x ; Relu(Relu(x)) = x
② { x: -ve → Relu(x) = 0 ; Relu(Relu(x)) = 0

from above 2 eqⁿ we say:

Relu(Relu(x)) = Relu(x)

* In Research it was found that when no. of layers are ↑ train and test loss was also ↑. So to get rid of this problem Resnet was introduced

* When we use Reguˡⁿ (l_1, l_2, dropout) it 0s the wts which are not important. and lets suppose in some layer all the wts become zero then it won't affect the network because of skip connection

* Let in fig ① 1st and 2nd conv layer is 0 then x' and x'' = 0 ⇒ x''' = 0 hence Relu(x) will be calculated.

* Adding additional / new layers would not hurt performance as reguˡⁿ will skip over them

* So if new layers are useful it will improve the performance of the model and if not useful it will be skipped

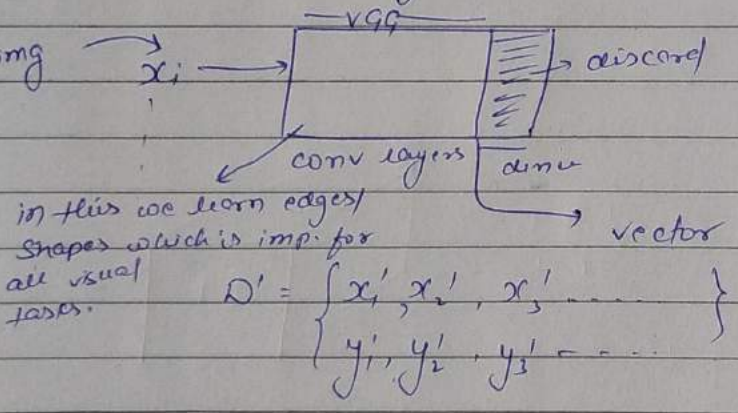
✱ Inception Network :- (refer ashukumar27.io/CNN-inception-Nw)

✱ instead of deciding whether to use 1×1 or 3×3 or 5×5 conv and whether to use max pool layer, why not use all of them?

✱ Transfer Learning :-

↳ idea :- Instead of building a CNN from scratch to solve our task, we can reuse existing models (VGG16) trained on diff dataset.

✱ **Case1** VGG16 + ImageNet



✱
- use case 1 when your dataset is very similar to imagenet and size of your dataset is small. (2000 img)
- use case2 when your dataset is similar to imagenet and large
- use case3 when your dataset is dissimilar to imageNet ~~(use initial layers)~~ and dataset is large
- ~~when dataset is small and~~ dissimilar use initial few layers of already trained model.

✱ **Case 2** we freeze ~~only~~ all the k conv. layer but last few layers. and fine tune last conv. layer using the new dataset by keeping the learning rate very small.

✱ **Case 3** use VGG16 + imageNet as initializat^n and train the whole model using new Dataset $D = \{x_i, y_i\}$ by keeping small learning rate.