Deep Learning Basics

Basic - Neuron

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

f - activation fn
w_i - weight
x_i - input

perceptron

a Neuron where,

$$f(x) = \begin{cases} 1, & \text{if } w^T x + b = 0 \\ 0, & \text{else} \end{cases} \qquad \sum_{i=1}^{n} x_i w_i + b$$

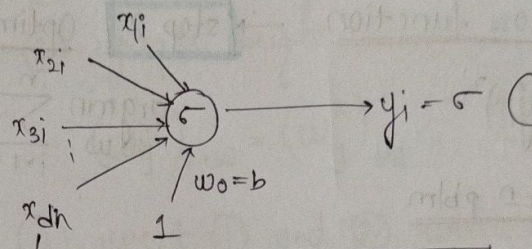
perceptron ≡ LR (diff: no squashing fn (σ))

How log-Reg ≡ Neuron

LR → $\hat{y}_i = \sigma(w^T x + b)$

In neuron,
let activation fn = σ

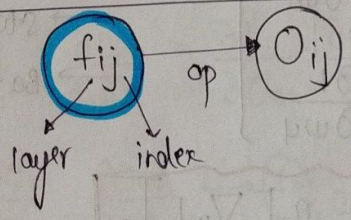
$$\hat{y}_i = \sigma\left(\sum_{i=1}^{n} w_i x_{ij} + b\right)$$

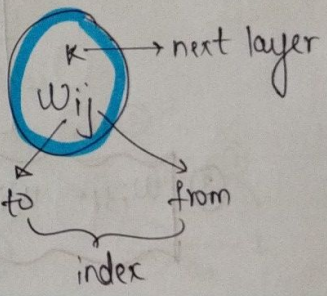


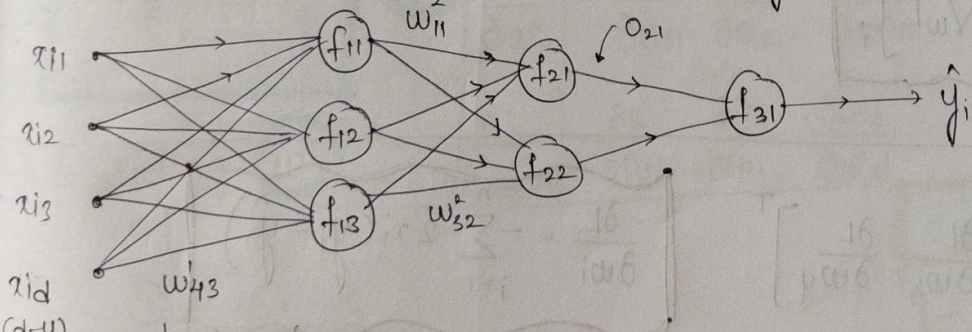
d = dimensions of data — last x_{di}

Notation

$D = \{x_i, y_i\}_1^n \quad x_i \in R^4 \quad y_i \in R^1$







$$w' \longrightarrow \begin{bmatrix} w'_{11} & w'_{12} & w'_{13} \\ w'_{21} & | & | \\ w'_{31} & | & | \\ w'_{41} & - & - \; w'_{43} \end{bmatrix}_{4\times 3}$$

similarly — $w' = \frac{\dim}{4 \times 3}$

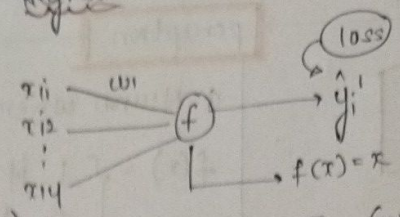
$w^2 = 3 \times 2$

$w^3 = 2 \times 1$

② **Train Single Neuron model** → find best weights.

classification — Log-reg / perceptron
Regression — linear regression

→ Linear regression as neural model



for Linear regression: $f(x)$ = identity fn

$\hat{y} = w^T x_i \; (w_1 x_1 + w_2 x_2 + \cdots + a+b+c)$

$\therefore \hat{y}_i = w^T x = (w_1 x_1 + w_2 x_2 + \cdots w_n x_n)$

loss fn → $\min_{w_i} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + reg$ we ignore regs for simplicity and $x_i \in \mathbb{R}^4$, $y_i \in \mathbb{R}^1$

→ **step 1** define loss function

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

→ **step 2** Optimization

$\underset{w}{\text{argmin}} \sum_{i=1}^{n} (y_i - f(w^T x))^2$

↳ Linear-R : identity
Log-R : sigmoid

→ **step 3** solve opt^n pblm

① Initialize w_i's, η

② $\nabla_w L = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \vdots \\ \frac{\partial L}{\partial w_4} \end{bmatrix}$

③ $w_{i+1} = w_i - \eta \left[\nabla_w L\right]_i$

Calculating $\nabla_w L$
→ Gradient descent — all pts (x_i, y_i)
→ stochastic GD — one pt
→ Batch-sep — some pts $< n$

Calculate $\nabla_w L$

$\nabla_w L = \left[\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial w_3}, \frac{\partial L}{\partial w_4}\right]^T$

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

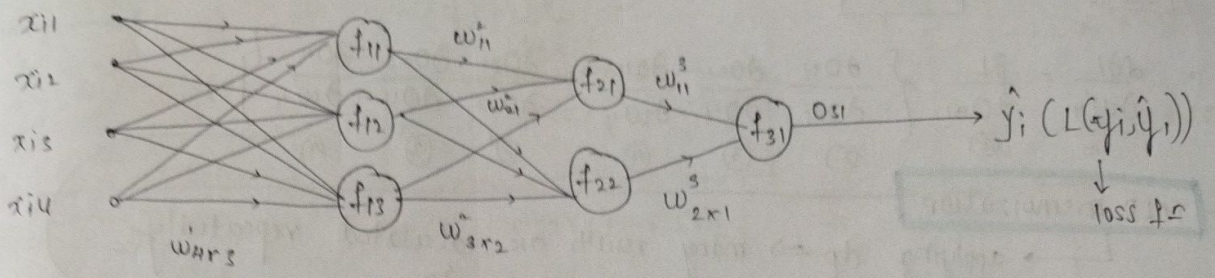
$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial w_1}$ using chain rule

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

$\frac{\partial f}{\partial w_1} = \frac{\partial w^T x_1}{\partial w_1} = x_1$

$\therefore \frac{\partial L}{\partial f} = -\sum 2 y_i - f(w^T x) \quad \frac{\partial f}{\partial w_i} = x_i$

Train multi-layer perceptron → task: find w^1, w^2, w^3 = 12+6+2 = 20 wts (3)



step 1 Loss fn

$$\text{Loss fn} = \sum (y_i - \hat{y}_i)^2 + \text{reg}$$

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

$$L = \sum_{i=1}^{n} L_i + \text{reg}$$

step 2

① Initialize weights, η

② update weights,

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

③ repeat ① and ② until convergence → Δw is small

optimization

$$\min_{w_{ij}^k} L$$

finding $\frac{\partial L}{\partial w_{ij}^k}$

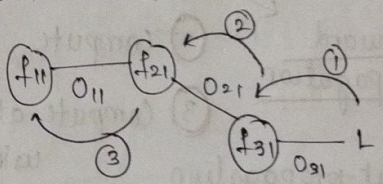
→ Calculating w^1

① find w^3

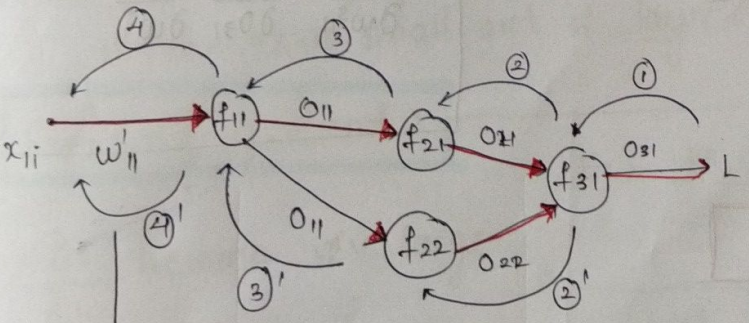
$$\frac{\partial L}{\partial w_{11}^3} = \frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial w_{11}^3} \quad ; \quad \frac{\partial L}{\partial w_{21}^3} = \frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial w_{21}^3}$$

w^2

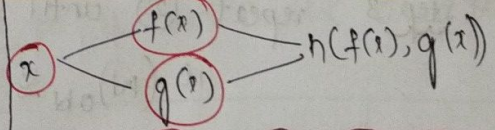
$$\frac{\partial L}{\partial w_{11}^2} = \frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial w_{11}^2}$$



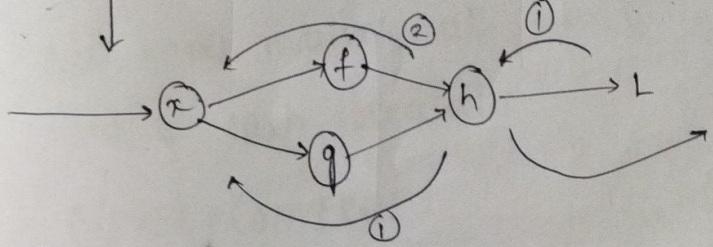
$$\frac{\partial L}{\partial w_{21}^2} = \frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial w_{21}^2}$$



Special Case of chain rule



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



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

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

④ $\frac{\partial L}{\partial w'_{11}} \cdot \frac{\partial L}{\partial o_{31}} \cdot \frac{\partial o_{31}}{\partial w'_{11}}$ → $\frac{\partial o_{31}}{\partial w'_{11}} = \left(\frac{\partial o_{31}}{\partial o_{21}} \cdot \frac{\partial o_{21}}{\partial o_{11}} \cdot \frac{\partial o_{11}}{\partial w'_{11}}\right) + \left(\frac{\partial o_{31}}{\partial o_{22}} \cdot \frac{\partial o_{22}}{\partial o_{11}} \cdot \frac{\partial o_{11}}{\partial w'_{11}}\right)$

$\therefore \frac{\partial L}{\partial w'_{11}} = \frac{\partial L}{\partial o_{31}} \left\{ \underset{②}{\frac{\partial o_{31}}{\partial o_{21}}} \cdot \underset{③}{\frac{\partial o_{21}}{\partial o_{11}}} \cdot \underset{④}{\frac{\partial o_{11}}{\partial w'_{11}}} + \underset{②'}{\frac{\partial o_{31}}{\partial o_{22}}} \cdot \underset{③'}{\frac{\partial o_{22}}{\partial o_{11}}} \cdot \underset{④'}{\frac{\partial o_{11}}{\partial w'_{11}}} \right\}$ (① = $\frac{\partial L}{\partial o_{31}}$)

MLP memoization

→ applying dp ⇒ many results are calculated repeatedly

Ex: $\frac{\partial o_{11}}{\partial w'_{11}}$

u can use chain rule

Back Propagation

Inp

* Activation fn r differentiable

Back-Prop = chain rule + memoization

$D = \{x_i, y_i\}$

Algorithm

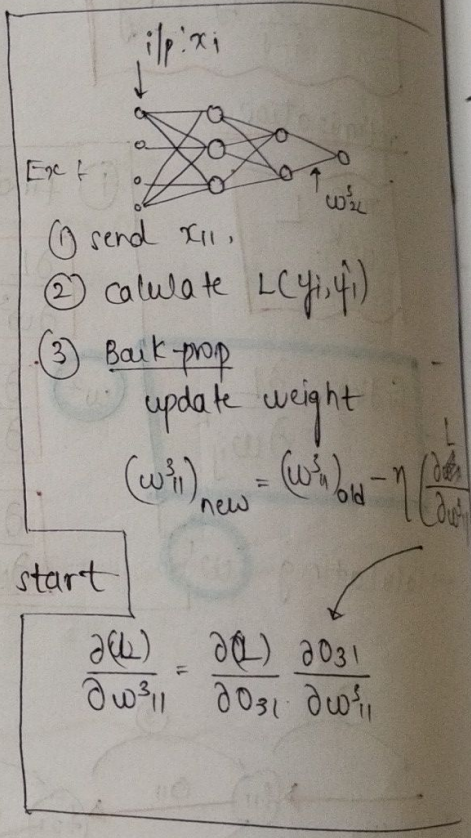
step 1 : Initialize W_{ij}^{K}'s

step 2 : for each (x_i) in D :

- ① pass x_i to forward n/w — forward propagation
- ② compute $L(y_i, \hat{y}_i)$
- ③ Compute all derivatives using chain-rule and memoization — Back-propagation
- ④ update weights from end of n/w to start

step 3 : repeat ② until convergence

ie $(W)_{old} \approx (W)_{new}$



if !convergence:

repeat algo for multiple epochs

mini-batch Back-prop

instead of sending (x_i) 1-by-1,

send them in batches of (K).

forward-prop

Activation Functions

Sigmoid

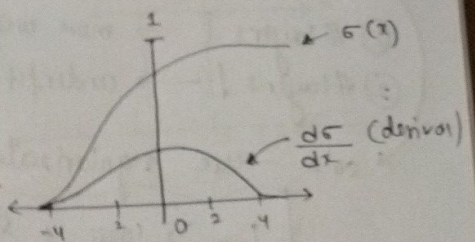
i/p : $w^T x = \sum w_i x_i = z$

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

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

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

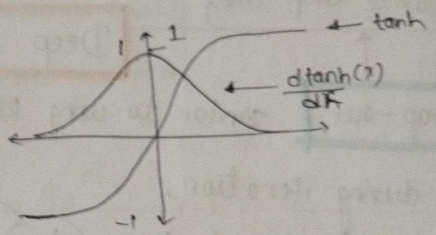
$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$



tanh

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

$$\frac{da}{dz} = 1 - a^2 \quad (a = \tanh(z))$$



sigmoid, tanh → old

ReLU, Leaky ReLU → new

Vanishing Gradients → slows downs convergence

$$w_{new} = w_{old} - \eta\left(\frac{\partial L}{\partial w}\right)$$

$$\frac{\partial L}{\partial w} = \left(\frac{\partial O_{31}}{\partial O_{21}} \frac{\partial O_{21}}{\partial O_{11}} \dots\right)$$

$$\frac{\partial O_{31}}{\partial O_{21}} = \frac{\partial \sigma(O_{21})}{\partial O_{21}}$$

wkt, $\frac{\partial \sigma}{\partial x} = 0 < x < 1$

each values are $0 \leftrightarrow 1$

→ when u multiply, the answer is very small $<<1$

as value of $\left(\frac{\partial L}{\partial w}\right)$ is v.small

$w_{old} - \left(\eta \cdot \frac{\partial L}{\partial w}\right)$ → negligible

∴ $w_{new} \approx w_{old}$ − v.v small value → so this cause $w_{new} \equiv w_{old}$ and it is assumed to converge

∴ due to vanishing grad, u can train small amt of layers $\leq 2, 3$, so u cant build deep NN.

Exploding gradients

Ex: $\frac{\partial L}{\partial w} = \left(\frac{\partial O_{31}}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial O_{11}} \dots\right)$

$>1 \quad \dots \quad$ = large number

(Not with sigmoid activation fn)

$\frac{\partial L}{\partial w}$ become v.v large

∴ Δw varies massively at end of each iterations

NEVER CONVERGES as $\Delta w \not\approx 0$

⑥ **Bias-variance tradeoff** – (Multi-layer perceptron)

① # layers ↑ → more weights → overfit / (high variance) ← L1 reg, L2 reg
② # layers ↓ → underfit (high-bias)

soln – use regularization

$L = \sum_{i=1}^{n} \text{loss}_i + \lambda \cdot L_2/L_1$ regularization on weights

L_1 → sparsity, some $w_i = 0$

regularize deep-NN

Deep Learning

Drop-out similar to using RF to act as regularization

* during iteration,
at each layer drop some neurons
↳ remove all connections

drop-out rate = p $(0 \le p \le 1)$

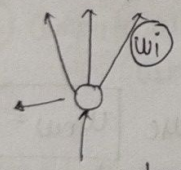
p = 0.2, 20% of neurons in layer are droped

→ in each iteration,
drop different neurons

similar to selecting some features in random-trees

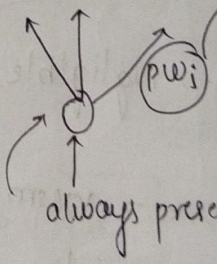
∴ dropout ≈ random subset of features/inputs

train time :-



neuron is present with probability = p = dropout rate

test-time :



why pwi ?

because during train 0 was present only p(p) time

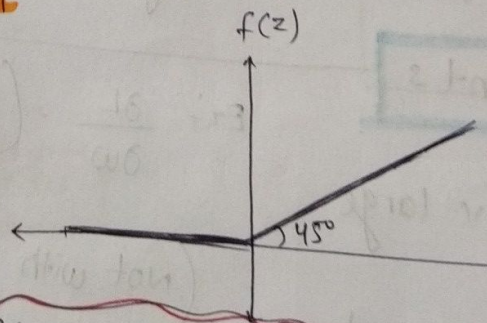
activation fn

ReLu

Rectified linear unit

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

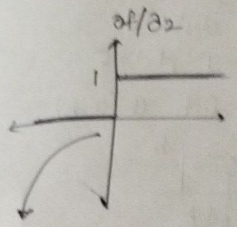
$$f(z) = \begin{cases} z, & \text{if } z > 0 \\ 0, & \text{otherwise} \end{cases}$$



$$\frac{df(z)}{dz} = \begin{cases} 0, & \text{if } z \le 0 \\ 1^*, & \text{otherwise} \end{cases}$$

→ tan(45) = 1

$\frac{\partial f(z)}{\partial z} = \begin{cases} 0, & z \le 0 \\ 1, & \text{otherwise} \end{cases}$



discontinous at $z=0$;
∴ not differentiable at 0

∴ $\frac{\partial f(z)}{\partial z} \in \{0, 1\}$

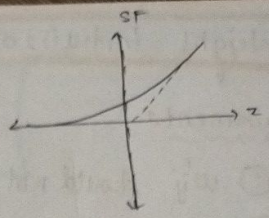
(i) never >1,
∴ !exploding gradient

(ii) cant be decimal like 0.001
∴ !vanishing gradient

(iii) if $\frac{\partial f(z)}{\partial z} = 0$, then all other terms dont matter $\Longrightarrow w_{new} = w_{old} - \eta(0 \cdots)$
∴ weights wont change
(Dead activation)

soln **softplus fn**

* $f(z) = \log(1 + e^z)$
* $f'(z) = \frac{1}{1 + e^{-z}}$



ReLu advantages

1) Speeds up convergence
2) Easy to compute derivatives - just if

disadvantage

1) dead activations → soln → Leaky relu / init strats
   ↳ when weights $\longrightarrow -\infty$

Variants of ReLu

① Noisy Relu

→ add gausian noise

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

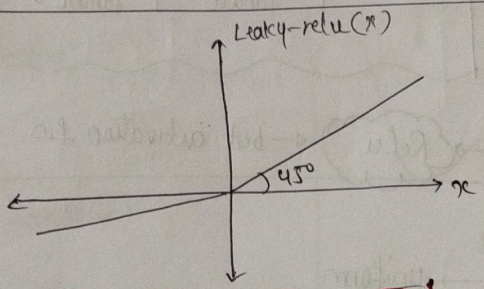
where,

$Y \sim N(0, \sigma(x))$

→ y is picked from a gausian distribution with small noise ≡ variance

② **Leaky ReLu**

→ solve dead activation problem



$\frac{df_{LR}}{dx} = \begin{cases} 1, & z > 0 \\ a, & z < 0 \end{cases}$

* this may lead to vanishing gradient because $0 < a < 1$ and if (a) is multiple times it may lead to vanishing

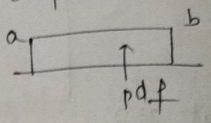
$f_{LR}(x) = \begin{cases} x, & x > 0 \\ a \cdot x, & x < 0 \end{cases}$

$a = 0.01$ → hyper param / fixed

⑧ Weight Initialization – picking weights for step ① of optimization

Requirements
① w_{ij}^{k} should not be zero
② w_{ij} should be small (! too small)
③ good variance $\text{var}(w_{ij}^{k})$
↳ more different
↳ better learning

* $N(0,\sigma)$
↳ normal dist^n
mean = 0, std = σ
* uniform dist $u(a,b)$



① Gaussian/Normal Init

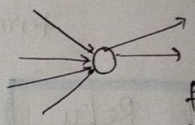
$$W_{ij}^{k} \sim N(0,\sigma)$$

pick — small var

② Uniform init → good for σ activat^n f^n

$$W_{ij}^{k} \sim u\left(\frac{-1}{\sqrt{fan\text{-}in}}, \frac{1}{\sqrt{fan\text{-}in}}\right)$$

fan-in : no of inputs
fan-out : no of outputs



Xavier/Glorot → sigmoid activ^n

→ gaussian

$$W_{ij}^{k} \sim N(0,\sigma_{ij})$$

$$\sigma_{ij} = \frac{2}{fanin + fanout}$$

→ unif

$$W_{ij}^{k} \sim u\left[\frac{-\sqrt{6}}{\sqrt{fanin + fanout}}, \frac{+\sqrt{6}}{\sqrt{fanin + fanout}}\right]$$

He-init^n ← Relu ← best activation f^n

→ Normal

$$W_{ij}^{k} \sim N(0,\sigma)$$

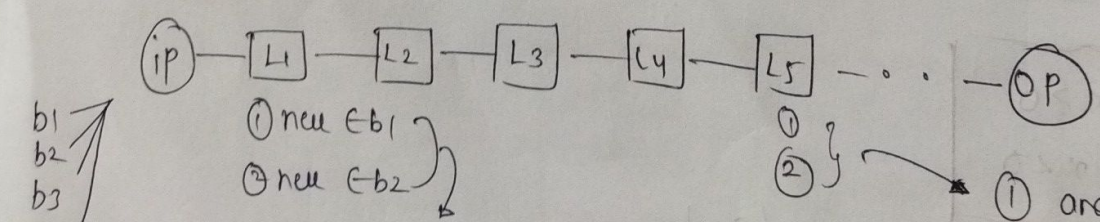
$$\sigma = \sqrt{\frac{2}{fanin}}$$

→ uniform

$$W_{ij}^{k} \sim u\left(-\sqrt{\frac{6}{fanin}}, +\sqrt{\frac{6}{fanin}}\right)$$

Batch-Normalization

→ consider ilps are normalized ie mean = 0, sd = σ
→ we send x_i as mini-batches



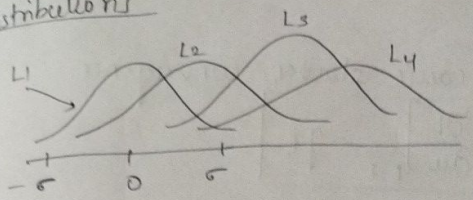
b1
b2
b3
b4

① new ∈ b1
② new ∈ b2

both are same as normalized

① ② } → ① and ② vary due to op^n performed by layers L2, L3,

distributions



Internal Covariance shifting

Can be avoided using batch normalization

∴ Deeper n/w's can be trained

soln → add a batch layer in b/w deep layer

... L5 → BN → L6 ...

↓ this normalizes data

Algorithm for Batch-Norm

→ advantages

BNL → y

① faster convergence

Input : values of x over minibatch

$B = \{x_1 \cdots m\}$

② works as weak regularization

use (BN + drop) always

output :- params to learn, γ, β → hyper params

$\{y_i = BN_{\gamma,\beta}(x_i)\}$

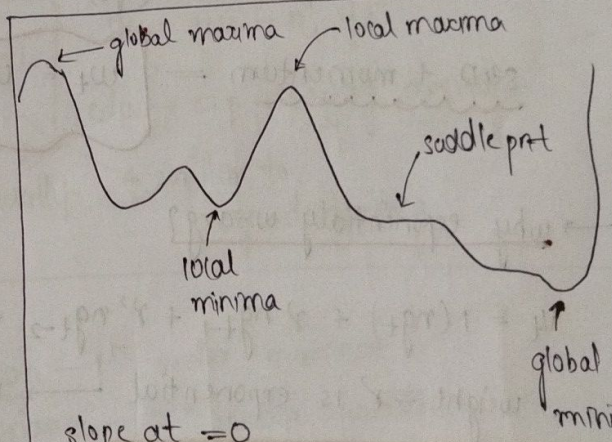
* $\mu_B \rightarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ (Batch-mean)

* $\sigma_B^2 \rightarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2$ (min-batch variance)

* $\hat{x_i} \rightarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ normalize

in case $\sigma_B^2 = 0$, dnr = ε very small value

o/p → $y_i = \gamma \hat{x_i} + \beta$ scale and shift



slope at = 0 minima

slope (saddle) = 0, ⟷ θ = 0°

tan θ = 0

problems

stuck at saddle point

(10) **SGD with momentum**

SGD ≈ GD, ∴ the gradients are noising which cause slower convergence

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

let $\left[\frac{\partial L}{\partial w}\right]_{t-1} = g_t$

* Denoise data

t_1 t_2 t_3 t_4 ...

→ a_1 a_2 a_3 a_4

data

$\gamma = 0 \leftrightarrow 1$

∴ $t_1 = a_1$

$t_2 = \gamma t_1 + a_1$

Exponenential weighted average

$V_1 = a_1$

$V_t = \gamma V_{t-1} + a_t$ → recursive fn

where, $0 < \gamma < 1$

$t_1 \rightarrow V_1 = a_1$

$t_2 \rightarrow V_2 = a_2 + \gamma V_1$

$t_3 \rightarrow V_3 = a_3 + (\gamma V_2)$

$= a_3 + \gamma(a_2 + \gamma V_1) = a_3 + \gamma a_2 + a_1\gamma^2$ → less weight to older points

more weight to current point

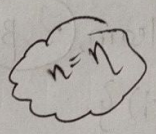
→ denoise SGD

momentum

initially, $V_1 = (g_1 \cdot \eta)$ ← learning rate

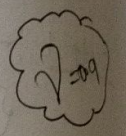
$V_t = \gamma V_{t-1} + (\eta g_t)$

$\Longrightarrow w_t = w_{t-1} - V_t$



$0 \leq \gamma \leq 1$

if $\gamma = 0$,

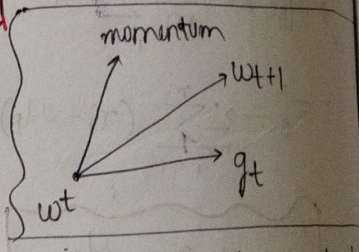


SGD + momentum = SGD

SGD + momentum = $w_t = w_{t-1} - [\gamma V_{t-1} + \eta g_t]$

exp-weigth avg (momentum)

gradient



→ why exponentialy w-avg?

$V_t = 1(\eta g_t) + \gamma \eta g_{t-1} + \gamma^2 \eta g_{t-2} \cdots$

weight ≡ γ is exponential → more weight to currently points and less value to points discoverd in past

Change LR (η) dynamicaly

ADAGRAD

→ In SGD the learning rate (η) is constant for all weights

idea have diferent (η) for diferent weight

why? when data has sparse features its helpful

Adagrad = adaptive gradient

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

η'_t is diferent fr each weight and each iteration

$$\eta'_t = \frac{\eta}{\sqrt{\alpha_{t-1} + \epsilon}}$$

$$\alpha_{t-1} = \sum_{i=1}^{t-1} g_i^2$$

→ sum of all (grodients)^2 upto t-1

$\downarrow$ $\left[\frac{\partial L}{\partial w}\right]_{w_{i-1}}$

always +ve

$\alpha_t \geq 0$ &
$\alpha_t > \alpha_{t-1}$

Note t – iteration number
as $t \uparrow$, $\alpha_{t-1} \uparrow \longrightarrow \eta'_t \downarrow$ ∴ as iterations increase, learning rate decreases adaptively

Uses
- → η tunes automatically
- → works if D has sparse and dense

problem? as $t \uparrow$, α_{t-1} may become v.v.Large
if $\alpha_{t-1} \to \infty$, $\eta'_t \to 0$ which causes slower convergence (very less Δ).

Adadelta & RMS prop → sol^n to (α_{t-1}) becoming v.v.large

Adadelta

$$\eta'_t = \frac{\eta}{\sqrt{eda_{t-1} + \epsilon}}$$

$$eda_{t-1} = \gamma \cdot eda_{t-2} + (1-\gamma) g^2_{t-1}$$

→ recursive

$$w_t = w_t - \eta'_t g_t$$

eda → exp decaying average

(γ) = 0.95 usually

→ similar to exp-decay-avg

uses:
→ control growth of eda_t

Adam (adaptive moment estimate)

In stats
mean – 1st order moment
variance – 2nd order moment

idea: along with using (g_t^2) for eda_t also use (g_t) → mean

→ variance

$$\hat{m}_t = \frac{\hat{m}_t}{1-(\beta_1)^t} \qquad \hat{V}_t = \frac{\hat{V}_t}{1-(\beta_2)^t}$$

→ $\beta_2 \wedge t$

eda g_t → $m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) g_t$

eda g_t^2 → $V_t = \beta_2 \cdot V_{t-1} + (1-\beta_2) g_t^2$

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

if $\beta_1 = 0$, adadelta (similar)

usualy, $\beta_1 = 0.95$ $\quad 0 \leq \beta_1, \beta_2 \leq 1$
$\beta_2 = 0.99$

(12) **Which optimizer to use?**

- mini-Batch SGD → gets stuck in saddle point → else good for smaller NN
- momentum & NAG → works well in most case but slower
- Adagrad, adadelta, rmsprop → good for sparse data
- Adam → better overall

Gradient monitor and clipping

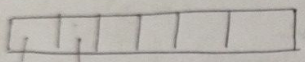
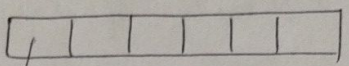
→ monitor gradients and updates for each epoch, layer and each weight

why? : vanishing gradient, exploding gradient

vanishing gradient → monitoring

exploding gradient → clipping

Grad clipping

$W =$  $g =$ 

w_{11} w_{12} $\frac{\partial L}{\partial w_{11}}$

→ W & g are vetor of all ~~vect~~ gradients and weights.

→ if weights dont change ⟹ vanishing gradient

clipping

$$\vec{g}_{new} = \frac{\vec{g}_{old}}{\|\vec{g}\|_2}$$

$$\|g\|_2 = \sqrt{g_1^2 + g_2^2 + \cdots g_n^2}$$

∴ in g_{new} each term ≤ 1

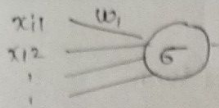
∴ it wont cause Δ exploding

Softmax Classifier → expanded upon on Logistic regression

→ Logistic regr? + multi class = Soft_max

*By defaut LR is for Binary class?, use 1 vs rest for multi class

Logistic regression as NN



$\rightarrow y_i = p(y_i = 1 | x_i)$

$\hat{y_i} = \sigma(z) = \frac{1}{1+e^{-z}}$, $z = w^T \cdot x$

$= \frac{e^z}{e^z+1}$

binary = y_i can be 0/1

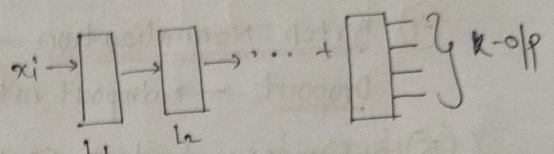
usage

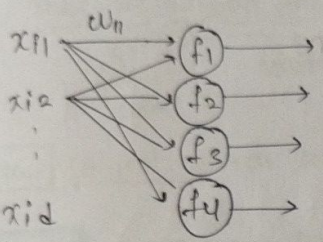
Softmax

$x_i \rightarrow$ [Model] $\rightarrow$ $p(y=1|x_i)$, $p(y=2|x_i)$, ..., $p(y=n|x_i)$ } sum = 1

$y \in \{1, .., n\}$

K-classfr — softmax layer





$\rightarrow$ let i/p to f_i be (z_i), $z_i = wx$

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

$\rightarrow f_i(x)$

$f_i(z_i) = \frac{e^{z_i}$ / $\sum_{i=1}^{K} e^{z_i}$

$f_1(z_1) = \frac{e^{z_1}}{\sum_{i=1}^{K} e^{z_i}}$ → sum of all e^{z_i}'s

where, K = no of f^n s / classes

sum of $f(z_i)$ ✓

$\frac{e^{z_1} + e^{z_2} + e^{z_3} + \cdots}{\sum e^{z_i}} = \frac{\sum e^{z_i}}{\sum e^{z_i}} = 1$

∴ LR → $f(z) = \frac{e^z}{e^z+1}$; softmax → $f(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$

LR → minimizes 2-class log-loss

softmax → minimizes multi-class log-loss

Cross entropy

multi-class log loss

N points, K classes

$$\text{mc-log loss} = \frac{-1}{N} \sum_{i=1}^{n} \sum_{j=1}^{K} y_{ij} \log(p_{ij})$$

→ (y_{ij}): binary indicator → {1, if $(y_i) \in$ class (j); 0, otherwise}

→ $(p_{ij}) = p(y_i == j | x_i)$

K = 2 → 2-class log loss

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

simple interpret

$$\frac{-1}{N} \sum_{i=1}^{N} \left(\text{does } (x_i) \in \text{class } j \times \text{prob}(x_i \in j) \right)$$

(14) **Steps to train MLP**

(1) Pre-process : data normalization

(2) weight initialization : . Xavier / glorot → sigmoid / tanh
He → ReLu
Gausian with small σ

(3) activation fn : ReLu

(4) Batch Normalization → deep layer (later layers)
Dropout → dropout rate (p)

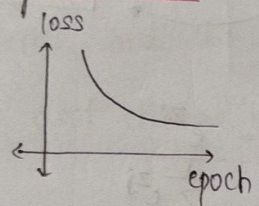
(5) optimizers : adam (2018) → fast to converge

(6) Hyper-params → architecture → #layers, #neurons
→ dropout rate (p)
→ Adam (B_1, B_2, α)

(7) Loss function : 2-class — log-loss
K-class — multiclass log-loss
regresion — squared error

(8) monitor gradients : clipping

(9) plots :



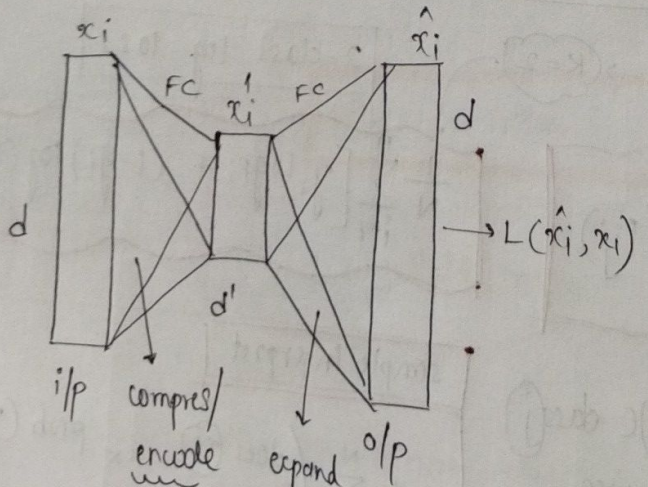
(10) ! avoid overfitting → very easy to overfit
↳ diff blw train / test loss

Auto Encoders

AE ≈ PCA — AE with 1 layer + f(·)

$x_i \to R^d \Rightarrow x_i' \to R^{d'}$

dimensionality reduction



lets consider, d = 6

→ hidden layer d' = 3

∴ 3 o/ps from hidden layer = x_i'

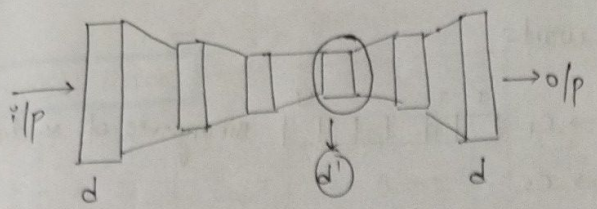
$x_i \to R^6$ $\hat{x}_i = R^6$

$x_i' \to R^3$

↓

dimensionality reduced

deep auto encoders



sparse autoencoder

→ use L1-regularization

$L(\hat{x}_i, x_i) = \|x_i - \hat{x}_i\|^2 + L_1$

↓ loss ↓ reg

$d' - \{x_i'\}^{d'}$ will become sparse

Word2Vec

- CBOW
- skipgram

$[x_i] \leftarrow D$

$V \rightarrow$ vocabulary / dictionary of words in D

v = no of words in V

Ex: The (cat) is on the wall

cat → focus word; is on the wall → context word

idea context words are useful in understanding focus words

→ given context words predict focus words

① CBOW — Continous - Bag of words

① construct V, $|V|_{len} = v$

② one-hot encode each word,

$w_i \in R^v$: binary vector of dim-v

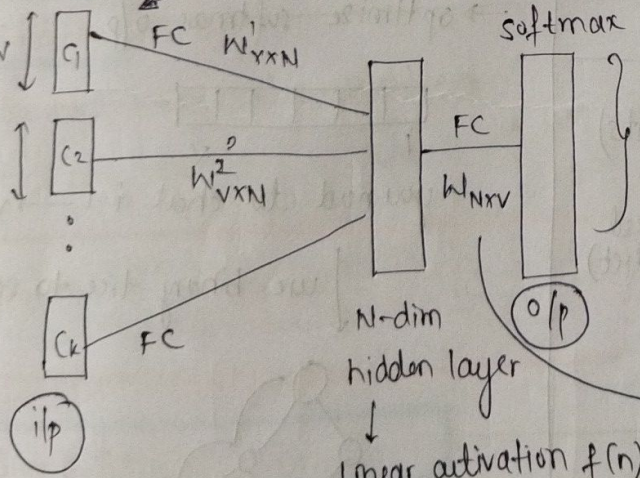
③ train a model

Train

① take all text

② make combination of FW | CW

Focus word	Context word
cat	{is, on, the, wall}
is	{the, cat, on, the, wall}
⋮	⋮

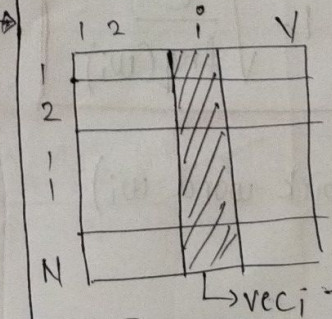


Linear activation $f(n)$ ie $f(x) = x$

v-dimensional binary vector

→ predict focus word $\in R^v$

End of training → W_{NXV}



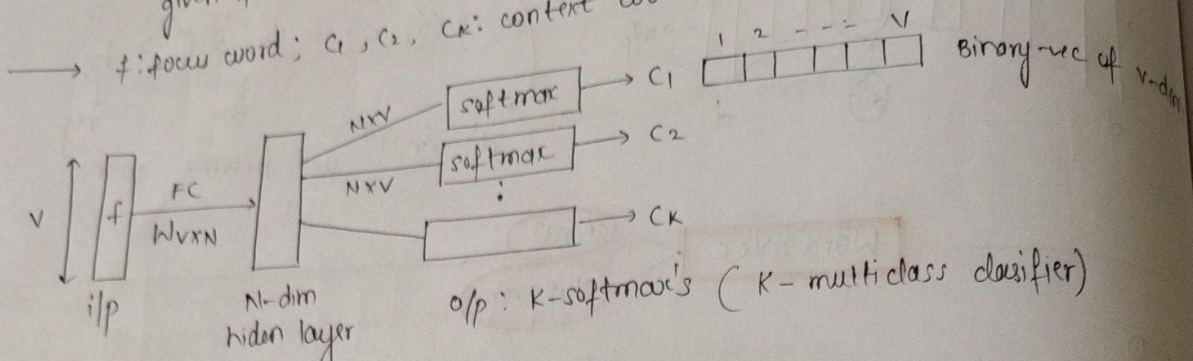
$w2v(w_i) = vec_i$ → N dimensional w2v

$vec_i \rightarrow w2v(word_i)$

(16) (2) **skip-gram**

given focus word predict context words.

→ f: focus word; $c_1, c_2, \ldots c_K$: context words



skipgram vs CBOW

weights in CBOW & skipgram = (K+1)(VXN) → VV large

CBOW → 1 softmax to train ∴ skipgram is computationaly expensive

skipgram → K softmax

CBOW pro :→ faster
→ beter for frequntly owning words

SKIP (+) smaller amt of data
(+) infrequntly owning words
} works well

(−) slower

optimizations for W2V due to large weights

- Heirarchical softmax (algo-based)
- Negative sampling (stats-based)

Neg-sampling

idea :- update only sample of words (weights)

(1) always keep target word (which u need to predict)

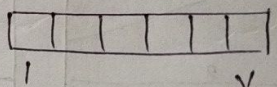
(2) sample from non-target words

$$P(w_i) = 1 - \sqrt{\frac{z}{freq(w_i)}} \rightarrow [10^{-5}]$$

→ P(you pick word w_i)

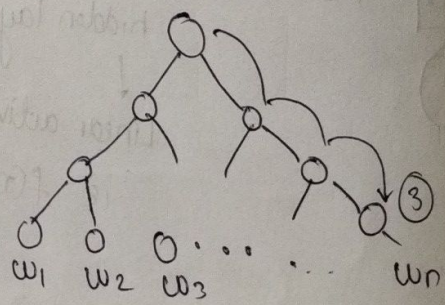
Heirarchical softmax

→ optimize softmax o/p.



→ you need to check i=1 → V

↓ use binary tree to search



incase (w_n) is answer (most probable)

u only need (3) iterations

Convolutional Neural Networks

Sobel Kernel

→ Edge detector

$$G_x = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 0 \\ -1 & -2 & -3 \end{bmatrix} \quad G_y = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 3 & 0 & -3 \end{bmatrix}$$

(detects edges)

convolution

$i/p_{6\times6} * G_{x\,3\times3} \rightarrow$ Image 4×4 with horizontal edges detected

$n\times n$ input $\xrightarrow{K\times K}$ $(n-k+1)\times(n-k+1)$ output

Padding

6×6 (n) $\rightarrow$ 4×4 ; 3×3 (k)

→ we want out shape ≡ input shape

$n=6,\ k=3$

→ $o/p = n-k+1 = 6-3+1 = 4$

we can get $o/p = 6$ if $n=8$, $8-3+1=6$

$2\times2 \xrightarrow{p=1} 4\times4$

add 2 col (R L) + 2 rows (T B)

→ padding with some value

$(n\times n) \xrightarrow[\text{pading}=1]{} (n+2)\times(n+2)$ → $(2\cdot p)$

Types of padding

1. Zero padding
2. Nearest pixel padding

$n\times n \xrightarrow[\text{pad}=p]{(K\times K)} (n+2p-k+1)\times(n+2p-k+1)$

padding + convolution

Stride

→ By how many values u shifted the conv-kernel (by default = 1)

→ if float, use floor ⌊ ⌋

$$n\times n \xrightarrow[\text{stride}=S]{K\times K} \left(\frac{n-K}{S}+1\right)\times\left(\frac{n-K}{S}+1\right)$$

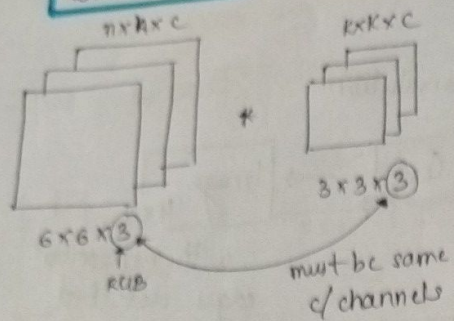
Padding + Stride + Conv ⌊ ⌋

$$(n\times n) \xrightarrow[\text{padding}=p,\ \text{stride}=S]{K\times K} \left(\frac{n-k+2p}{S}+1\right)\times\left(\frac{n-k+2p}{S}+1\right)$$

Input

(18)

Conv on RGB

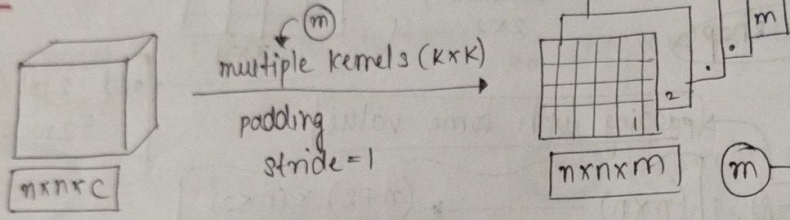


$\longrightarrow (n-k+1) \times (n-k+1) \times 1$ → o/p is always a matrix/ 2D o/p

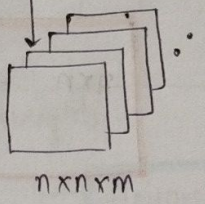
Convolution layer in NN

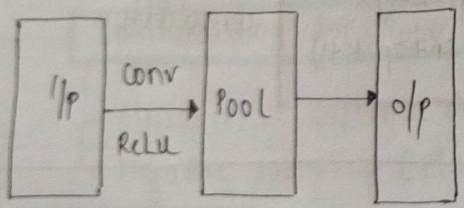
MLP → learn weights

CNN → learn Kernel matrices



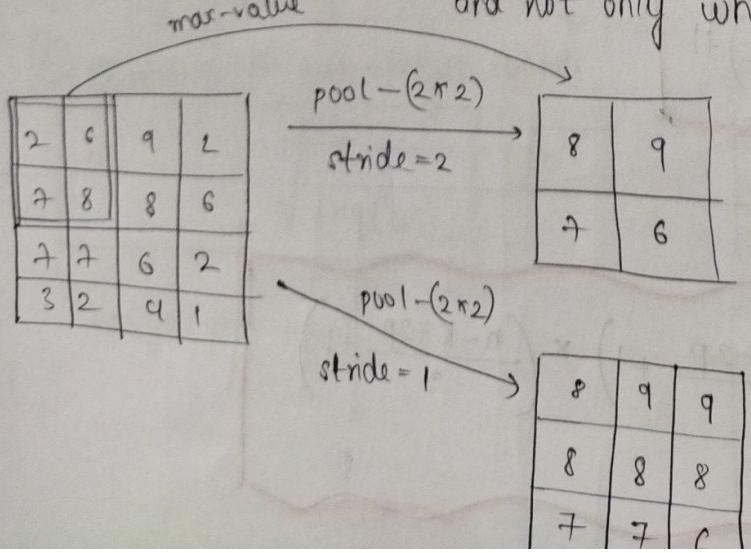
apply element-wise activation (RELU)





max-pooling → same as conv → but pick max value

↳ aim: location invariance :- the feature could be present anywhere in image and not only where it was seen



we can say if val↑, then is atleast one occurr of feature in image

CNN Training

* In backprop, wrt loss fn must be differentiable and all operations

condition
- 1) conv layer ✓
- 2) max-pool layer

→ must be differentiable

conv + Relu → differentiable
- → element wise mul + addition
- → similar to dot product $w^T x$ → is differentiable

derivative of max-pooling

pool → $f(x) = \max(x)$
- → [elements which occur in max pool kernel]

	0	1	2
0	6	2	9
1	1	3	4
2	5	6	7

mp = 2×2 →

6	9
6	67

6 depends only on (2,1)

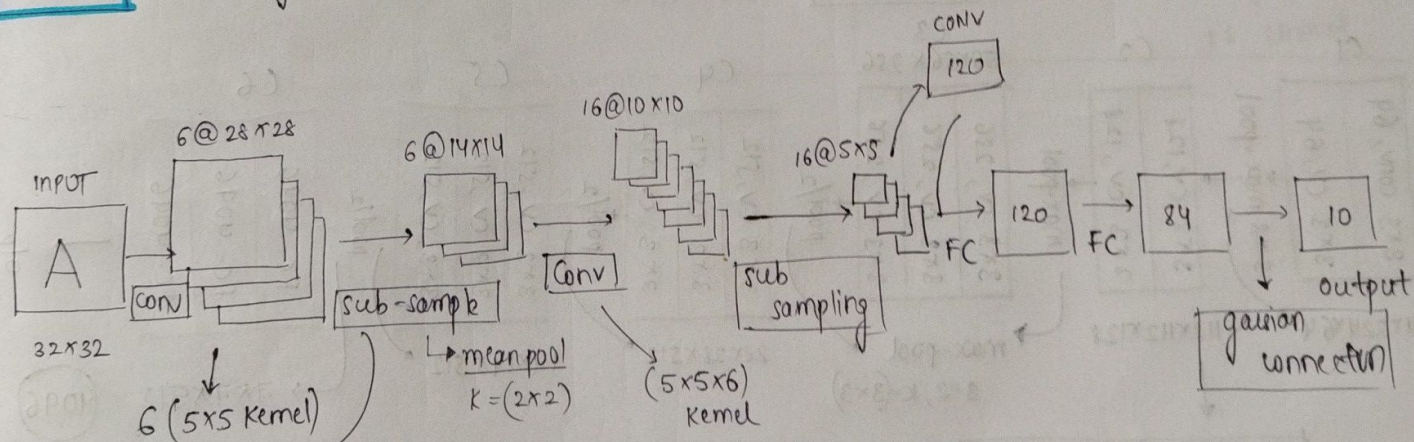
$\therefore \frac{\partial mp}{\partial x_i} = 1$ for (x_{21})

for others $\frac{\partial mp}{\partial x_i} = 0$ bcoz mp doesn't depend on x_i

$f'(x) = g(x) = \begin{cases} 1, & \text{if } x = \max(x) \\ 0, & \text{else} \end{cases}$

$\left(\frac{df(x)}{dx_i}\right)$

LeNet → sigmoid activation



Sub-Sampling ≡ Mean Pooling

1	2
3	4

→ 10/4

Data Augmentation

→ modify same data to make different variations

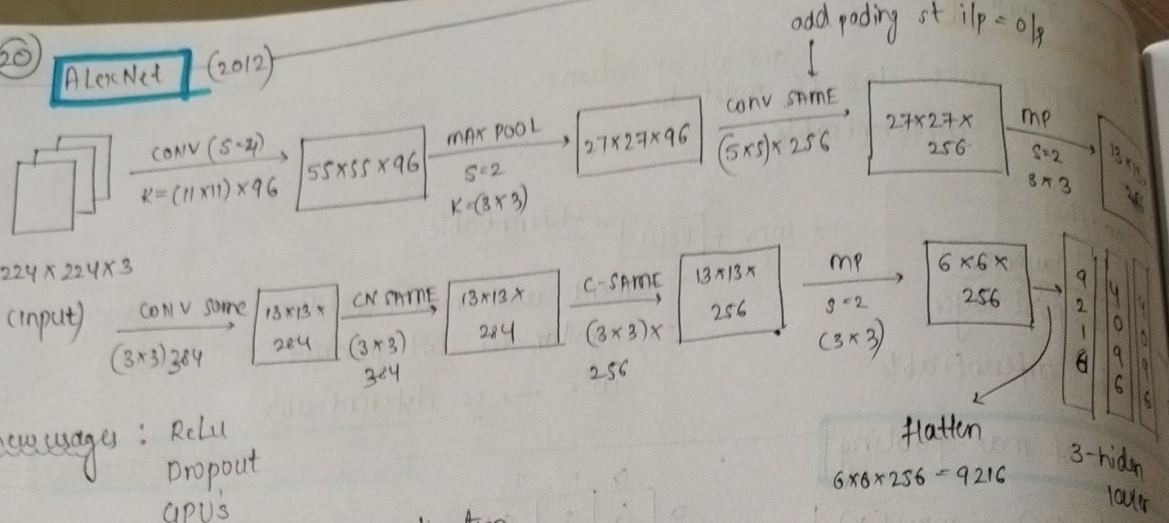
→ rotate, flip, skew, noise, reshape ..etc, zoom, shape

Use

→ Reduces various invariances.

→ convert small dataset to large dataset

(20) **AlexNet** (2012)



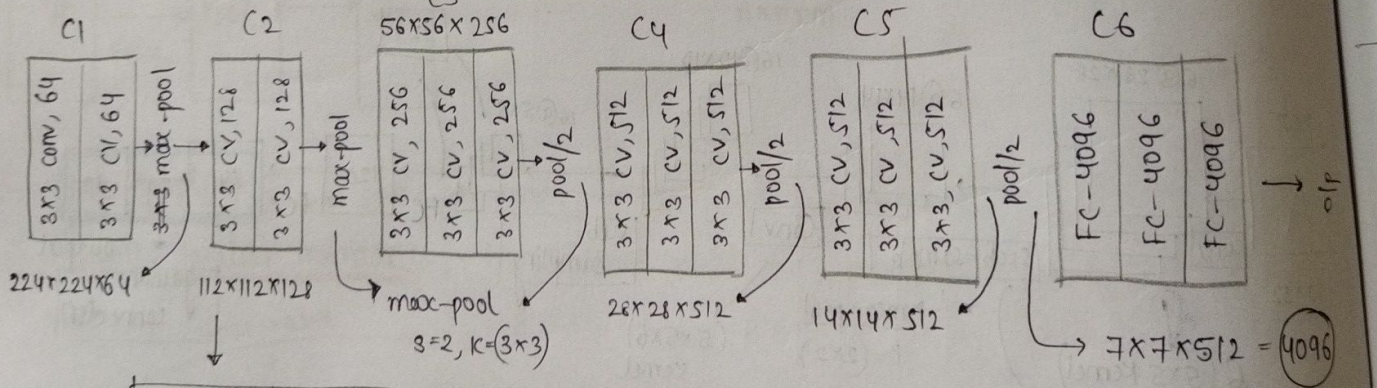
new usages : ReLU
Dropout
GPUs
→ Local Response Normalization (LRN)

VGG Net → simplified improoved version of alexnet

convolutions (conv) → (3x3) S=1 p=same and (max-pool) → (2x2) (S=2)

- VGG-16
- VGG-19

Convnet architecture of **VGG-16**



output dim after (C2) completes

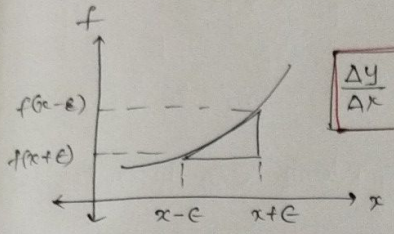
i/p (224x224x3) → o/p (4096) → 1000 classes

ResNets | Residual Networks

Gradient Checking

* Numerical approximation of derivative of f = $\lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x-\varepsilon)}{2\varepsilon}$

* Actual der → calculated by formula or backprop.



$\frac{\Delta y}{\Delta x} = \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$

because ϵ is small, difference also is small

$$\text{Grad\_check} = \frac{\|dw - dw_{approx}\|_2}{\|dw\|_2 + \|dw_{approx}\|_2}$$ → Euclidean distance

→ vvsmall $< 10e^{-7}$ → correct

$> 10e^{-3}$ → false / not correct

Ex: $W = w_1, w_2$, $X = x_1, x_2$ | $f(w_1, w_2, x_1, x_2) = w_1^2 x_1 + w_2^2 x_2$

$\frac{df}{dw_1} = 2 \cdot w_1 x_1 \rightarrow (1)$ $\frac{df}{dw_2} = 2 \cdot w_2 x_2$; $\left(\frac{df}{dw_1}\right)_{approx} = \frac{f(w_1+\varepsilon_1, w_2 \cdot) - f(w_1 - \varepsilon_1, ,,)}{2\epsilon}$

$(dw) = 2 w_1 x_1 = 6 \rightarrow (1)$

$(dw_1)_{approx} = 5.9999999 \rightarrow (2)$

Grad_check = $\left(\frac{6 - 5.9999\cdots}{6 + 5.9999}\right) = 4.251268e^{-13}$

∴ as $< e^{-13}$ → backprop is correct