So, L2 reg. doesn't change the value of w_1 from one iteration to another.

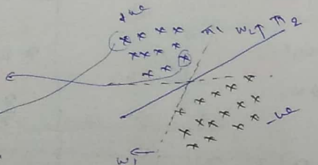
, L1 reg. continues to constantly reduce w_1 towards $w_i^* = 0$.

↳ So, chance that ($w_1 = 0$) are more at the end of k-iterations

Support-Vector-Machine (SVM)

Geometric Intuition:

SVM is a popular ML algo. used in classification / Regression, developed back in 1990's



for this point $P(y=1) \approx 0.5$

→ prob. is approx 0.5

for this point prob that $y=1$ is approx 0.9.

→ i have many πs that separate the +ve from -ve pts are possible.

key idea of SVM :- Choose π that separates the +ve from -ve pts as widely as possible.

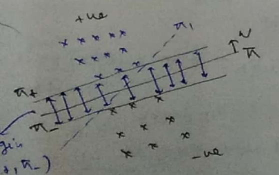
Here π is better than π'

→ π is margin maximising hyperplane

is parallel to π

is parallel to π

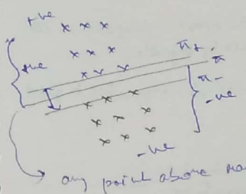
and π_- is also parallel.



to find a plane π that maximizes the margin = dist(π^+, π^-)

because, margin ↑, generalization accuracy ↑

Here, when margin will increase. $P(y=1)$ will ... So, acc will increase.

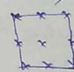


points through which π^+ or π^- pass through are called SVs (Support vectors)

→ any point above margin maximum hyperplane is +ve and below it is -ve.

⇒ Alternative Geometrical intuition of SVM :-

one way to do it using Convex hull → It is a closed path that each point should lie inside it.

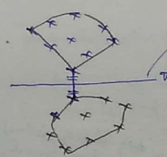


→ this is not a convex hull because, line joining two point is outer for some line.

① Create Convex hull for +ve pts and -ve pts.

② find the shortest line connecting these hulls.

③ Bisect the line to get margin max^m hyperplane.



Mathematical Derivation

π = margin-maximization.

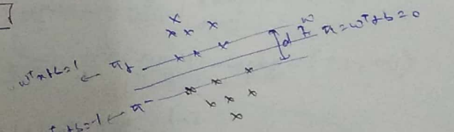
let :- π : $w^Tx + b = 0$

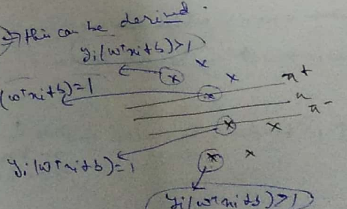
if π_+ : $w^Tx + b = 1$

π_- : $w^Tx + b = -1$

we have, margin (d) = $d = \frac{2}{||w||}$ ⇒ this can be derived.

$(w^*, b^*) = \underset{w, b}{\text{argmax}} \frac{2}{||w||}$ = margin





Constraints of this optimization problem of SVM.

$w^*, b^* = \underset{w,b}{\operatorname{argmax}} \frac{2}{\|w\|}$

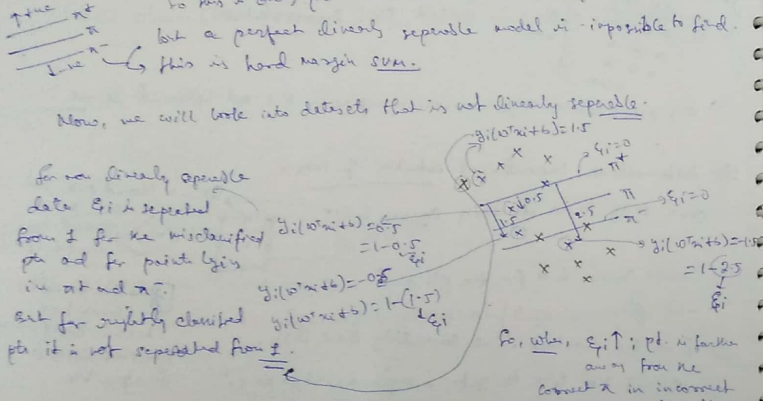
s.t. $\forall i,\ y_i(w^T x_i + b) \geq 1$

So this is only possible when data is linearly separable.

but a perfect linearly separable model is impossible to find.

↳ this is hard margin SVM.

Now, we will take into datasets that is not linearly separable.



for non linearly separable data ξ_i is separated from 1 for the misclassified pts and for points lying in or not on π^-.

But for suitably classified pts it is not separated from 1.

So, when $\xi_i \uparrow$; pt. is farther away from the correct π in incorrect direction.

$x_i \to \xi_i$

$\xi_i = 0$ if $y_i(w^T x_i + b) \geq 1$

↳ Correctly classified pt.

$\xi_i > 0$ & it is equal to some unit of dist away from the correct hyperplane in the incorrect dir.

Aim

$(w^*, b^*) = \underset{w,b}{\operatorname{argmax}} \left(\frac{2}{\|w\|}\right) = \underset{w,b}{\operatorname{argmin}} \left(\frac{\|w\|}{2}\right)$ → avg. dist of misclassified pts from correct plane

$\max f(\cdot) = \min \frac{1}{f(\cdot)}$

$(w^*, b^*) = \underset{w,b}{\operatorname{argmin}} \left(\frac{\|w\|}{2}\right) + C \cdot \frac{1}{n} \sum_{i=1}^{n} \xi_i$ → $\frac{1}{\text{margin}}$

s.t. $y_i(w^T x_i + b) \geq 1 - \xi_i \ \forall i$ ↳ for correctly classified points $\xi_i = 0$

$\xi_i \geq 0$

minimum error = minimum misclassification = $\min \sum \xi_i$

↳ this is in general regularization term.

loss.

on $C \uparrow$; tendency to make mistake ↓ on Dtrain

⇒ overfit ⇒ high variance

$C \downarrow$; underfit ⇒ high bias

As in logistic-regression ⇒ $\lambda \uparrow$ ⇒ high bias

$\lambda \downarrow$ ⇒ high var

$C = \frac{1}{\lambda}$

Why we take value +1 and −1 for support vector planes.

Here just for convenience we take 1 and −1, we can also take any the number $+k$ and $-k$

↳ But more value planes to have the same, we need a proper bisector of plane.

$w^T x + b = 1$

$\pi: w^T x + b = 0$

$\pi^-: w^T x + b = -1$

$\|w\| \neq 1$

↳ w can be any vector it need not to be a unit vector.

Since, our main goal is to maximize the margin

$\therefore w^*, b^* = \underset{w,b}{\operatorname{argmax}} \frac{2}{\|w\|} = \underset{w,b}{\operatorname{argmax}} \frac{2k}{\|w\|} = \frac{2}{\|w\|}$ if $k = 1$

↳ Then value will be same for both.

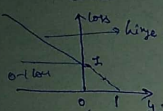
(2) $w^T x + b = k$

$\left(\frac{w}{k}\right)^T x + \left(\frac{b}{k}\right) = 1$ → Since k is a scalar no. so, it can't have such value can be easily change, no 1.

$(w')^T x + b' = 1$

Loss Function (Hinge loss) based interpretation.

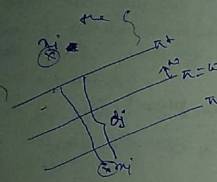
The core nature of loss in SVM is hinge type.



SVM – hinge loss + regularization.

hinge loss :- $z_i \geq 1$; hinge-loss = 0
$z_i < 1$; hinge-loss = $1 - z_i$
↳ $\max(0, 1-z_i)$

when $z_i > 0$: x_i is correctly classified
$z_i < 0$: x_i is incorrectly classified.



Case 1 :- when $z_i \geq 1$
↳ $1 - z_i$ is -ve value
then, $\max(0, 1-z_i) = 0$

Case 2 :- $z_i < 1$; $1 - z_i > 0$
↳ $\max(0, 1-z_i) = 1 - z_i$

$d_j = 1 - y_j(w^T x_j + b) = 1 - z_j$

& ξ_j = dist from x_j to the $x^+ = d_j = 1 - z_j$
↳ when x_j is misclassified.

Since, x_j is misclassified from the region so, distance is taken from the.

$\xi_i = 0 \Rightarrow$ Since x_i is correctly classified.

Soft SVM :- $\min_{w,b} \frac{\|w\|}{2} + C \sum_{i=1}^{n} \xi_i$ → loss.
→ neglecting $\frac{1}{2}$ here because it is constant for a data set.

C↑ → overfit
C↓ → underfit

s.t. $(1 - y_i(w^T x_i + b)) \geq \xi_i \quad \forall i$
$\xi_i \geq 0$

Loss-min :- $\min_{w,b} \sum_{i=1}^{n} \max(0, 1 - y_i(w^T x_i + b)) + \lambda \|w\|^2$
↳ regularization.
z_i

λ↑ → underfit
λ↓ → overfit

when $\|w\| \geq 0 \Rightarrow$ min value of $\frac{\|w\|}{2}$ is zero or $\min \|w\|^2$

So, Soft SVM is similar as loss-min.

Dual Form of SVM Formulation

Suppose if Sonar data given as similarity matrix, then in soft-margin SVM it is not interpretable, so in it is converted into its equivalent Dual Form, and most often dual form is used.

Soft margin SVM :-
$\min_{w,b} \frac{1}{2}\|w\| + C \sum_{i=1}^{n} \xi_i$
s.t. $y_i(w^T x_i + b) \geq 1 - \xi_i \; \forall i$
$\xi_i \geq 0$
↳ this is also known as Primal of SVM.

equivalent

$\max \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j (x_i^T x_j)$
s.t $C \geq \alpha_i \geq 0$
$\sum_{i=1}^{n} \alpha_i y_i = 0$

this is also known as similarity matrix and stays together

Our main aim of Soft margin SVM is to find $f(x_q) = (w^T x_q + b)$
$\Downarrow$
y_q

So, dual form will also calculate $f(x_q)$
① for every x_i there is a corresponding value α_i ($x_i \to \alpha_i$)
② x_i's only occur in the form of $(x_i^T x_j)$
③ $f(x_q) = \sum_{i=1}^{n} \alpha_i y_i x_i^T x_q + b$

For Support vectors (SVs) :- $\alpha_i > 0$
for non SVs :- $\alpha_i = 0$

So, for $f(x_q) \Rightarrow$ only point that matters is SVs, if we add any non-SVs points, it won't effect $f(x_q)$

$\Rightarrow \max_{\alpha_i} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j x_i^T x_j$

s.t $\alpha_i \geq 0 \Rightarrow$ for SVs $\alpha_i = 0 \Rightarrow$ SVs
$\sum_{i=1}^{n} \alpha_i y_i = 0$ $\alpha_i > 0 \Rightarrow$ non SVs

So, $x_i^T x_j = x_i \cdot x_j = \text{Cosine-Similarity}(x_i, x_j)$ → if $\|x_i\| = 1$; $\|x_j\| = 1$
↳ Cosine Angle of

So, $(x_i^T x_j) \Rightarrow$ Can be thought of any similarity
$k(x_i, x_j)$
↳ k = kernel function.

So, during run time.

$$f(x_q) = \sum_{i=1}^{n} \alpha_i y_i (x_i^T x_q) + b$$

↳ $k(x_i, x_q)$

$$\hookrightarrow \sum_{i=1}^{n} \alpha_i y_i k(x_i, x_q) + b$$

Kernel Trick

till now we saw that $x_i^T x_j$ can be changed to any similarity kernel function.

↳ $k(x_i, x_j)$

↳ the most important idea in SVM is kernel Trick.

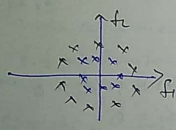
Since, Soft SVM-hyperplanes can ~~also~~ only be applied to linearly separable data.

But kernel function or feature transformation is logistic regression, transform non-linear data and separate them using hyperplanes.

The only difference b/w logistic regression and SVM till now is SVM uses margin-maps.

linear SVM $= x_i^T x_j$

(kernel SVM $\Rightarrow k(x_i, x_j)$)



hyperplane

- Low SVM - fail to separate
- log-reg $\rightarrow$ fail
- log-reg + feature transform ↳ succeed
- kernel SVM - succeed with right kernel.

So, kernelization in SVM handle non-linearly separable dataset.

Polynomial kernel

this dataset is not linearly separable so, kernelization is used.

or we use feature transformation in log-regression $(f_1, f_2) \xrightarrow{FT} (f_1^2, f_2^2)$ ↓ log-reg.

$k(x_1, x_2) = (x_1^T x_2 + c)^d$ → d can be any number.

eg: $k(x_1, x_2) = (1 + x_1^T x_2)^2$

Quadratic kernel

$$k(x_1, x_2) = (x_1^T x_2)^2 \longrightarrow x_1 = \langle x_{11}, x_{12} \rangle \Leftarrow \text{2D data}$$

$$x_2 = \langle x_{21}, x_{22} \rangle$$

$$= (1 + x_{11}x_{21} + x_{12}x_{22})^2$$

$$= 1 + x_{11}^2 x_{21}^2 + x_{12}^2 x_{22}^2 + 2x_{11}x_{21} + 2x_{12}x_{22} + 2x_{11}x_{21}x_{12}x_{22}$$

Let $[1, x_{11}^2, x_{12}^2, \sqrt{2}x_{11}, \sqrt{2}x_{12}, \sqrt{2}x_{11}x_{12}] : x_1'$ (taking 2nd term)

$[1, x_{21}^2, x_{22}^2, \sqrt{2}x_{21}, \sqrt{2}x_{22}, \sqrt{2}x_{21}x_{22}] : x_2'$ (taking 2nd term)

↳ 6-Dimension.

$= (x_1')^T (x_2')$

So, during kernelization: (d) $\xrightarrow{FT}$ (d') internally implicitly (d' > d)

Kernel 2D $\xrightarrow{FT}$ 6D.

So, kernelization is an internal process.

Mercer's Theorem:

↳ It shows how using kernels, a non-separable data can be converted into linearly separable.

d → d'

RBF Kernel → Radial Basis function.

SVM is the most popular/general-purpose: RBF

$$K_{RBF}(x_1, x_2) = \exp\left(-\frac{\|x_1 - x_2\|^2}{2\sigma^2}\right)$$

↳ hyperparameter.

d_{12}: x_1 — x_2 → $\|x_1 - x_2\|^2 = d_{12}^2$

In RBF-marginal kernel SVM σ is also a hyperparameter.

↳ $k(x_1, x_2) = \exp\left(\frac{-d_{12}^2}{2\sigma^2}\right)$ → these distances are related to similarity.

when d_{12} ↑ ; $k(x_1 x_2)$ ↓ similarity ↓

@ $\frac{1}{\sigma^2}$ ↓ : d_{12} ↑, d^2 ↑

@ d^2 ↑ $\frac{1}{2\sigma^2}$ ↓

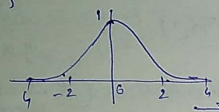
So, $k(x_1, x_2) > k(x_1, x_3)$

So, Similarity b/w x_1 and x_2 are greater than Similarity b/w x_1 and x_3.

⇒ Here, Similarity depends on σ. and σ in RBF SVM, are similar to k in kernel.

Suppose, we have. $\sigma = \{1, 0.01, 10\}$

for $\sigma = 1$ $\exp(-x^2/2)$ →



So, we can see that

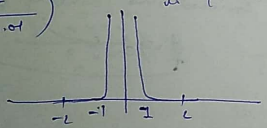
when $d \uparrow \rightarrow \exp(\frac{-x^2}{2 \cdot 1}) \rightarrow$ reaches to 0

So, here, when dist > 4. ↳ $\exp(\frac{-x^2}{2 \cdot 1}) = 0$.

$\exp(\frac{-x^2}{2 \cdot 1}) = 1 \rightarrow$ when $d = 0$.

So, the Similarity b/w the distance exist here when dist in b/w -2 to 2.

for $\sigma = 0.01$ $\exp(\frac{-x^2}{2 \times 0.01})$



So, $\sigma = 0.1$

$\sigma^2 = 0.01$

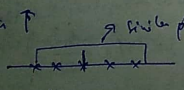
when dist > 1, $k = 0$.

So, here, the Similarity dies b/w the dist -1 to 1.

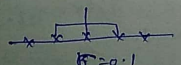
So, we can see that $\sigma = 10$

when $\sigma \uparrow$, Similarity ↑ $\sigma^2 = 100$

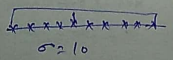
Since in kernel when $k \uparrow$ the range of Similar points ↑



$\sigma = 1$



$\sigma = 0.1$



$\sigma = 10$

So, ($\sigma \uparrow$ in RBF) ⇒ $k \uparrow$ in Kernel

But in k-nn we had memory issue $O(nd)$

But in RBF SVM ↳ it only depends on Support vectors, which is smaller than all points in knn.

So, it can be easily executed to get $f(x_q)$ ↳ class corresponding to any query points.

So, when we have no any idea about which kernel to use, then use RBF SVM ↳ it will work good.

C, σ ⇒ can be set by grid search, Random Search.

Domain Specific kernels.

When we have domain knowledge + Feature Transformation → kernel-trick.

Like in text classification → ~~kernel~~ string kernel is used.

So, we should have a good domain knowledge.

Train and Runtime Complexity of SVM.

Best library for training SVM's is LibSVM, but we can use sklearn anyway.

Training is done by sequential minimal optimization (SMO)

Runtime Complexity of SVM is → $O(n^2)$ for kernel-SVM's.

↳ so for large dataset, SVM can never be used. Since internal kernel operations

$$f(x_q) = \sum_{i=1}^{n} \alpha_i y_i K(x_i, x_q) + b$$

$\alpha_i = 0$ for non SVs.

for support vectors (SVs) SVs = k → $1 \le k \le n$

↳ $O(kd)$

nu-SVM Control and support vectors.

In C-SVM → that is the original form, we have $C > 0$

But we have an alternative formulation of SVM

i.e. nu-SVM, where $0 \leq \nu \leq 1$

Hyperparameter

ν → fraction of errors

ν ≤ fraction of SVs

SVs ≥ 1% of n

if n = 100,000 SVs ≥ 1000

But this much support vectors could lead to large time complexity.

Suppose we are training our data with SVM algo. and error can't be more than 1%. Then, $\nu = 0.01$

"SVM Regression"

As SVM is used for classification purpose, it is also used for Regression problems.

$$\min_{w,b} \frac{1}{2}\|w\|^2$$

$$\text{s.t.y } y_i - (w^T x_i + b) \leq \varepsilon + \xi_i$$

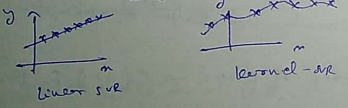
$$(w^T x_i + b) - y_i \leq \varepsilon + \xi_i$$

ξ_i → hyperparameter

This can also be kernelized.

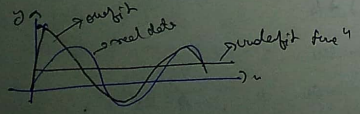
to fit in non linear data also.

$\varepsilon > 0$



error ↓ ← $\varepsilon \downarrow$ ⇒ errors are low in training data but overfitting ↑

$\varepsilon \uparrow$ ⇒ errors in Data ↑ ⇒ underfit ↑.



Cases

→ In SVM instead of doing feature Engineering & Feature Transform; we find the right kernel, or "RBF"

→ Decision Surface: when we have lr. SVM → hyperplane

kernel SVM is used in case of non-linear surface

→ When we have similarity func's / distance func's

↳ kernel SVM will benefit from it $k(x_i, x_i')$

→ SVM can't do good Interpretability and can't find a feature importance.

→ SVM is very less affected by outliers since it depends on support vectors only.

↳ Bias-Variance: $C \uparrow$ ⇒ overfit ⇒ high-Variance

$C \downarrow$ ⇒ underfit ⇒ high-bias

Can be found by grid search.

σ (RBF) ↓ low

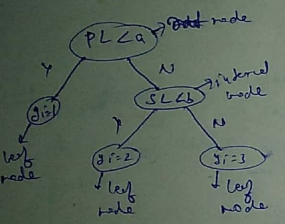
Decision Trees

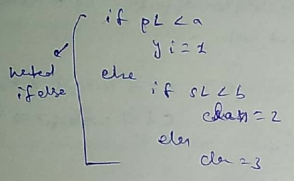
Geometric Intuition of decision tree Axis parallel hyperplanes

Decision trees → it is simply a nested if else statement.

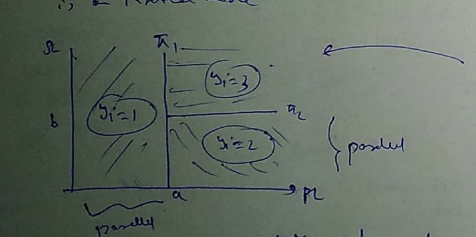
algos: Naive Bayes (probabilistic method), log. regr, lin. regr, SVM → kernel. (geometric, hyperplanes)

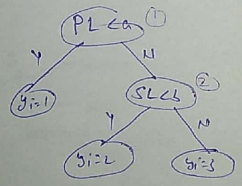
As, i- Iris dataset, are pairs, $x_i = \langle PL, PW, SW, PW \rangle$





nodes and vertex are same, any node that is not a root node or a leaf node is an internal node





all these hyperplanes are axis's parallel in a Decision Tree.

Sample Decision Tree

↳ Refer pdf provided in Lec 2 of Module 2

Building a Decision Tree (DT) Entropy

Entropy is often used in information theory & electronics and physics.

Suppose, we have random variable

$Y \rightarrow y_1, y_2, y_3, \ldots y_k$

Entropy: $H(Y) = -\sum_{i=1}^{k} P(y_i) \log_b (P(y_i))$

b can be b=2 or b=e=2.71

$\log_2 = \lg$

$\log_e = \ln$

$P(y_i) = P(Y = y_i)$

Suppose we have a dataset where there is probability that the person will play tennis or not

if $P(Y_i = \text{Yes}) = \frac{9}{14}$ play tennis

& $P(y_i = \text{No}) = \frac{5}{14}$ not play tennis

Then Entropy:

$$H(Y) = -\frac{9}{14} \lg\left(\frac{9}{14}\right) - \frac{5}{14} \lg\left(\frac{5}{14}\right)$$

$$= 0.94$$

Properties:- $Y = Y_+, Y_-$ (2-class, 2 category)

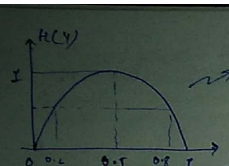
Case 1:- $D \rightarrow y_+ \rightarrow 99\%$, $y_- \rightarrow 1\%$ ⟹ $H(Y) = -0.99 \lg(0.99) - 0.01 \lg(0.01)$ $= -0.0807$

Case 2:- $D \rightarrow y_+ \rightarrow 50\%$, $y_- \rightarrow 50\%$ ⟹ $H(Y) = -0.5 \lg(0.5) - 0.5 \lg(0.5)$ $= 1$

Case 3:- $D \rightarrow y_+ \rightarrow 0\%$, $y_- \rightarrow 100\%$ ⟹ $H(Y) = 0$.

So, when classes are equally probable then Entropy is maximum. Here, at 50%.

Entropy decreases when there is dissimilarity in class.



when classes are equally probable entropy is max^m.

$-0.2\log 0.2 - 0.8\log 0.8$
$-0.8\log 0.8 - 0.2\log 0.2$

$y_+ \to 0.2$ $y_+ \to 0.8$
$y_- \to 0.8$ $y_- \to 0.2$

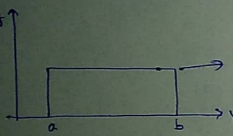
save entropy.

$P(y_+) = 1 - P(y_-)$

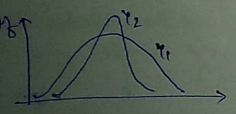
So, $Y \to y_1\ y_2\ y_3 \cdots y_k$

↳ equally probable → entropy is max^m

if $Y_1 \to$ most probable
$Y_2 Y_3 \cdots Y_k \to$ less probable $= 0$ → entropy is min^m.



uniform distribution through out
↳ entropy is max^m.

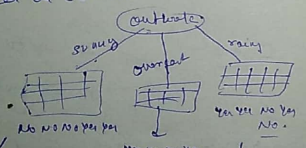


$Y_1 \to$ less peaked
$Y_2 \to$ very peaked
$H(Y_2) < H(Y_1)$

Building a decision Tree Formation Train

Suppose we have made our decision tree based on outlook.

We previously have calculated the entropy for outlook $H_D(\text{outlook}) = 0.94$



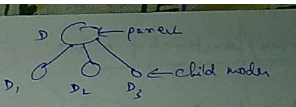
$H_{D_1}(Y) = 0.97$
entropy for play tennis or not

$H_{D_2}(Y) = 0$

$H_{D_3}(Y) = 0.97$

Information Gain
$IG(Y, \text{outlook}) = \left(\frac{5}{14} \times 0.97 + \frac{4}{14} \times 0 + \frac{5}{14} \times 0.97\right) - 0.94$
weighted entropy after D_1, D_2, D_3.
$= \left(\frac{5}{7} \times 0.97 - 0.94\right) = 16$

So, we have



$$I_H = \left(\frac{|D_1|}{|D|} H_{D_1}(Y) + \frac{|D_2|}{|D|} \times H_{D_2}(Y) + \frac{|D_3|}{|D|} \times H_{D_3}(Y)\right) - H_D(Y)$$

total points in D_1 / total pts in D

Building a decision tree Gini Impurity.

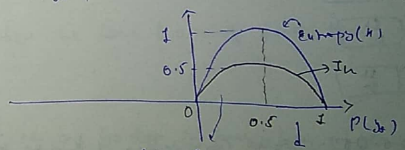
This is similar to entropy. But ~~extra~~ calculation of entropy involves log function and calculation of Gini Impurity involves square term. And we know that log takes more computation time than square. So, Gini Impurity is most often used in sklearn.

Gini Impurity $\hookrightarrow I_G(Y) = 1 - \sum_{i=1}^{k} (P(y_i))^2$

$Y \to Y_1, Y_2, Y_3 \cdots Y_k$

Case 1 :- $P(y_+) = 0.5$
$P(y_-) = 0.5$
$I_H(Y) = 1 - (0.25 + 0.25)$
$= 0.5$
$H(Y) = 1$

Case 2 :- $P(y_+) = 1$
$P(y_-) = 0$
$I_H(Y) = 1 - (1 + 0)$
$H(Y) \to = 0$



$P(y_+) \uparrow$ $H \uparrow$ $I_H \uparrow$
$P(y_+) \uparrow$ $H \downarrow$ $I_H \downarrow$
→ So entropy and Gini Impurity behaves similar.

$I_H = 1 - (P(y_+)^2 + P(y_-)^2)$

$H(y) = -P(y_+)\log(P(y_+)) - P(y_-)\log((P(y_-))$

Building a decision tree / Constructing a DT

Suppose we have a dataset with 9 (+ve pts) and 5 (-ve pts), So,

Entropy. $H(Y) = 0.94$.

That dataset is further divided on basis of (Outlook).

D → Outlook (D)

Sunny | Overcast | Rainy

D_1 (2+, 3-) | D_2 (4+, 0-) | D_3 (3+, 2-)

$HD_1(Y)$ | $HD_2(Y)$ | $HD_3(Y)$ → weighted entropy = 0.69.

(Entropy based on dataset D_1, D_2, D_3)

So, Information Gain IG = (0.25) = (0.94 − 0.69)

Dataset D can also be divided on basis of (temperature)

H (2+, 2-) | M (4+, 2-) | C (3+, 1-)

Similarly, Humidity: H (3+, 3-) | N (6+, 1-)

Windy: F (6+, 2-) | T (3+, 3-)

D →

$IG(Y, \text{Outlook}) = 0.25$. provides correspondingly to this we will have for Information Gain (IG) of all dataset,

$IG(Y, \text{temp}) = $ —

$IG(Y, \text{Humidity}) = $ —

$IG(Y, \text{Wind}) = $ —

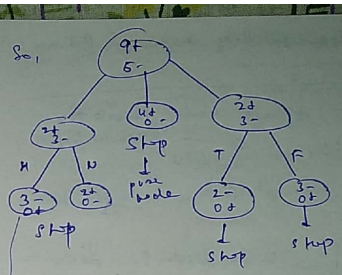
So, we will choose that as (root) (node) which has highest IG.

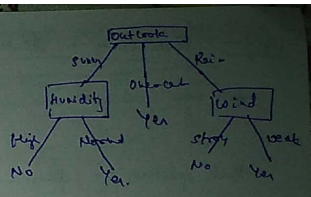
$IG(Y, f)$ = entropy at parent level − weighted entropy at child level.

parent →

child →

So,





So, recursively bracketing each node using IG as the criterion.

So, after parent node is decided, to decide the next level node again, IG is used.

once, we reach at a point when we have all the same point in a node left → like (3+, 0-) → straight opinion.

then we stop going further.

So, we will stop making decision tree, when,

① pure node → stop growing the node

② when we have bucket of points then stop growing node

③ if we are too deep in tree, then we can stop growing nodes

So, then, after going through a lot of if else condition,

Suppose visually we are (h = 10 k) at last point found = 2

this is noise, this can be ignored because considering this we are overfit.

→ depth of tree ↑ ; overfit ↑ (few points)

depth is small ⇒ underfit

DT - hyperparameter = depth

Can be found by - Cross validation.

Building a decision Tree splitting numerical features.

till now we have discussed about discrete random variable or Categorical feature. this was easy to split.

But splitting the dataset according to numerical feature is a bit time consuming.

Suppose we have to split

f	y
2.2	1
2.6	1
3.5	0
3.8	0
4.6	1
5.3	0

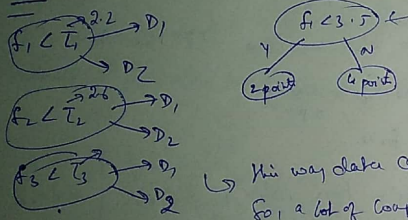
→ So for splitting this in decision tree.

① Sort the numerical features in ascending order.

② $f_1 < 2.2$
$f_1 < 2.6$
⋮
$f_1 < 4.6$
$f_1 < 5.3$

Here we have to split data w.r.t every point.

e.g.:-



↳ this way data can be splitted into n ways.

So, a lot of computation is required.

So, once we did it, we take the feature with most information gain or value.

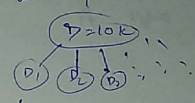
Feature Standardization.

Since decision tree is not distance based so unlike KNN, SVM, we doesn't do feature standardization or feature normalization here.

Here in DT each value of feature is taken as threshold and if we have a sorted it not, DT only care about it.

Building a decision Tree Categorical feature with many possible values

Suppose, we have Zipcode/Pincode as feature then at this point making a DT w.r.t pincode is not good because this will be a lot of sparsity



zipcode	y (0,1)
P_1	1
P_2	0
P_3	1
⋮	0

So, if we convert these zipcode into categorical even this won't solve the problem.

So, one hack is change zipcode into numerical feature as it is

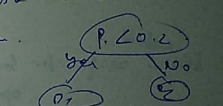
Suppose, $P(y_i = 1 | P_j) = \frac{19}{20}$

↓

prob. of $y_i = 1$ given zipcode $= P_j$

this will convert zipcode into corresponding numerical value

So, Now, we can split and do further process.



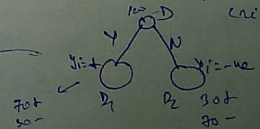
Now these two datasets have a large number of value.

Overfitting and Underfitting

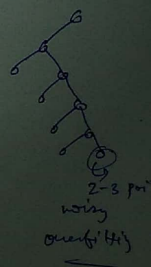
when depth of tree increases (↑) → possibility of very few points at leaf nodes increases. ↳ Interpretability of model also decreases. because there will be a lot of if else condition

depth ↑ → overfitting increases.

when, depth = 1 = decision stump (like umpire in cricket)



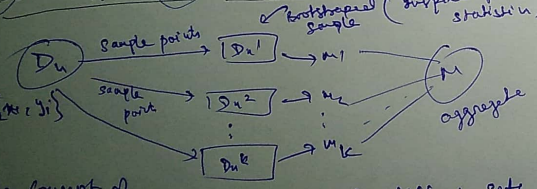
this is not true.



Ensemble learning.

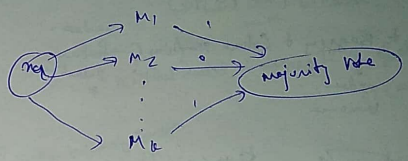
↳ It is like combining different sets of model and finally gives an output A/T combined model.

Bootstrapped Aggregation (Bagging) Intuition.



Here, the concept of bagging is → Sample D_n will be different sets with replacement, then train the sets with different models. and then finally aggregate the model.

Here, each model M_i has seen a different subset of data.



Suppose when we have binary classification, when different model predicts different value, we take the majority vote value. when, it is a regression prob. we take the mean or median of all the predicted output by diffrnt models.

Bootstrapping (Bagging) is used to reduce Variance.

Here, if we reduce the the number of samples from train data then it won't effect much.

So, Bagging can reduce Variance in a model without impacting the bias

model error = $Bias^2 + Var$ → reduced!

if base model (M_i) = low bias, high var model → reduced by bagging.

So, in DT of depth (large)

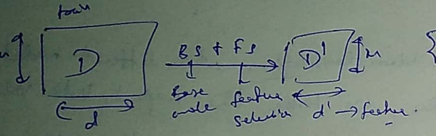
↳ high variance, low bias → Can be fixed by bagging

RF

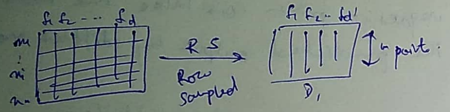
Random forest and their Construction :-

↳ Random forest is also a bootstrap sampling,

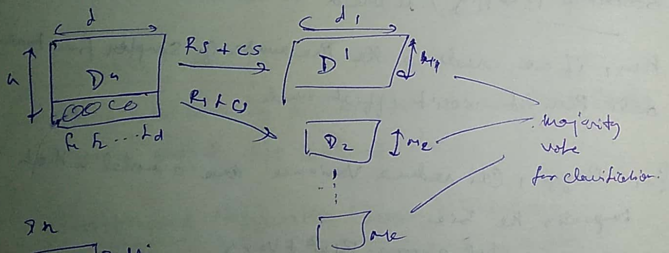
Random Forest = DT + Bagging + Column sampling {→ random feature selection
(DT ↙ Base model; Bagging ↓ reduce variance)

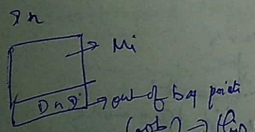


{ m < n
d' < d



Here, when random forest is applied if train model in reduced random features and rows.





→ out of bag points (oob) ⇒ this can be used for cross validation.

Bias-Variance tradeoff

As discussed RF reduces variance.
↳ has low bias because base learners (M_i's) are low bias.

actual model is aggregate of all the models.

$M = agg(M_1, M_2, M_3 \dots M_k)$

Since, bias of discrete models (M_i) is small
∴ bias of actual model is also small

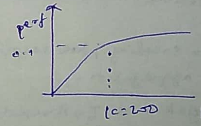
$bias(M) = bias(M_i)$

{ as, k↑ ; variance↓
k↓ ; variance↑

$D_n \xrightarrow[CS\ (80\%)]{RS\ (20\%)} D'$

Row sampling ratio : $\frac{m}{n}$ = row S.R.
Column sampling ratio : $\frac{d'}{d}$ = Col. S.R.

↙ when we sample row and column
lets say for 20% data and 40% data, then, there is very less chance that model will change when training data changes ∴
Col SR ↓ → low var model
row SR ↓ → var ↓



k = # base learner

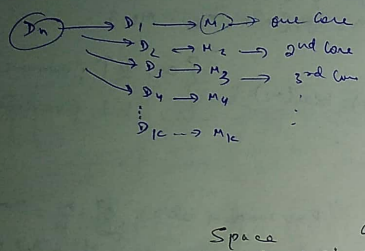
→ Here, when k↑, learning rate will increase, but after some particular value of k → it learning gets fixed,

So, row SR, col S.R and k ⇒ can be calculated by Grid Search.

Train and Runtime Complexity.

RF with k-base learners (DT)

Since most computers have different cores, so, RF is designed to perform parallely on individual core.



training time complexity
$= O(n \log d \times k)$

Run time complexity
$O(\text{depth} \times k)$

Space Complexity $= O(DT \times k)$ → plainly if else code.

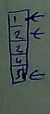
Extremely randomized trees

↳ It is somewhat similar to Random Tree.

↳ ExtraTreeClassifier ⇒ ~~from~~ sklearn.model_selection
ExtraTreeRegressor. ⇒ from sklearn.ensemble import ExtraTreeClassifier.

In RF we have seen that, there is Column sampling, row sampling, and aggregation.

and it targets all possible values of f_i to determine τ → threshold
↳ gini

Best f_i →  ← Checks for every threshold

Best in Extreme Trees: There is Column sampling + row sampling + aggregation + randomization when selecting "τ"

it selects threshold randomly.

reduces variance better than RF.

Random Tree cases.

↳ DT, RF → don't handle large feature dimension.

In DT → bias variance trade-off depended on depth.
But here on RF → it can be controlled by k = # base learners.

For feature importance

DT → It shows feature importance by overall reduction in Entropy & Gini Entropy (f_i) became of only one feature in DT. (model)

But in RF → overall reduction in Entropy became of feature variance (each of M_i's).

Boosting Intuition.

In Bagging we have high variance, low bias base model + randomization + aggregation ($\langle S, R \rangle$) → reduce variance.

But in Boosting, we have low variance, high bias + additively combine. → reduces bias while keeping on low variance.

Core idea → reduce bias.

high bias → high train error.
↳ eg: DT which has 2 or 3 depth.

$D_{train} = \{x_i, y_i\}_{i=1}^{n}$

we first train a model, then we get some errors

(a) $D_{train} \rightarrow$ Model 1 $\rightarrow$ some train error.
$(x_i, y_i)_{i=1}^{n}$ ↳ $y = h_0(x)$

(b) $y_i - h_0(x_i) = \text{error}_i$ → simple difference error

Step 2 ↳ train again a model with feature x_i and previous error.

$M_1 \leftarrow \{x_i, \text{error}_i\}_{i=1}^{n}$
$h_1(x)$ $\quad \text{error}_2 = y_i - h_0(x_i)$

model of end stage 1 = $F_1(x) = \alpha_0 h_0(x) + \alpha_1 h_1(x)$

↳ weighted sum of two base models.

③ $\{x_i, \text{error}_i\} \rightarrow (M_2) \; (h_2(x))$

↳ $y_i - F_1(x_i)$

$F_2(x) = \alpha_0 h_0(x) + \alpha_2 h_2(x) + \alpha_1 h_1(x)$

3rd model

So, at the end of stage k.

$F_k(x) = \sum_{i=0}^{n} \alpha_i h_i(x)$ → trained to fit the residual error at the end of previous stage

↳ additive weight model

$h_i(x) \leftarrow \{x_i, \text{error}\}$ → $y_i - f_{i-1}(x)$

↳ residual error at end of stage $(i-1)$

So, with reducing training error in each stage, we end up reducing loss residual & train error So, bias ↓

Gradient Boosted DT (GBDT) is also used in interweb Coupons.

Residuals, Loss function and Gradients

$F_k(x) = \sum_{i=0}^{k} \alpha_i h_i(x)$ here, $h_i(x)$ depends on residual error.

residual at end of stage k → $\text{error}_i = y_i - F_k(x)$

residual

Loss minimization can be done by different ways for different model.

Logistic-loss ← classifn

Linear Regn ← Sq. loss ← Visualization of this is easier So, we will take this.

SVM ← hinge loss.

↳ $L(y_i, F_k(x_i)) = [y_i - F_k(x_i)]^2$ {let $f_k(x_i) = z_i$}

Sq. loss, $\therefore \frac{\partial L}{\partial F_k(x_i)} = \frac{\partial L}{\partial z_i} = \frac{\partial}{\partial z_i}(y_i - z_i)^2$

$= (-1) * 2 * (y_i - z_i)$

$\frac{\partial L}{\partial F_k(x_i)} = -2(y_i - z_i)$

$\frac{-\partial L}{\partial F_k(x_i)} = 2(y_i - z_i)$

-ve derivative. | +ve | residual

So, negative gradient ≃ residual ↳ pseudo-residual

(So, residual can be replaced by -ve gradient)

So, pseudo residual lets us have any loss function which is differentiable

So, Gradient Boosting can minimize any loss which is differentiable.

But models like RF can only minimize hinge loss.

So, $(x_i, \text{error}_i) \rightarrow (M_i \; h_i(x_i))$

↳ pseudo-residual $\frac{-\partial L}{\partial F_{i-1}(x)}$

Gradient Boosting

Input training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, f(z))$

no. of iteration M.

Algorithm (Gradient boosting)

at first γ is mean of y_i

$m = k$

① Initialize model with a constant value.

zeroth model ← $F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma)$ ↳ constant γ: $\arg\min \sum_{i=1}^{n} (y_i - \gamma)^2$, $\gamma = \text{mean}(y_i)$

② From $m = 1$ to M

(1) → Compute so-called pseudo-residuals.

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x) = F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n$$

(2) Fit a base learner (e.g. tree) $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$

(3) Compute multiplier γ_m by solving the following one dimensional optimization problem.

$$\gamma_m = \arg\min \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + \gamma h_m(x_i))$$

(4) Update the model :-

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

③ Compute $F_M(x)$

Regularization by shrinking.

In gradient boosting there is a very high chance of overfitting a model.

So, by shrinking we saves the model from overfit.

$$F_m(x) = h_0(x) + \sum_{m=1}^{M} \gamma_m h_m(x)$$

base-model ↑ → overfit ↑ → variance ↑

So, Shrinkage is added

$$F_m(x) = F_{m-1}(x) + \nu \cdot \gamma_m h_m(x)$$ where $0 < \nu \leq 1$

↳ learning rate.

Since ν is very small. So, $\nu \gamma_m h_m(x)$ ↳ is also small so effect of this on model won't increase the variance.

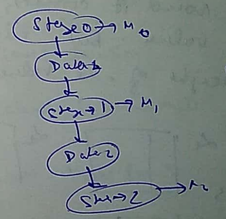
Then, γ and ν can be searched by Gridsearch using CV.

when M is large ↳ model is more likely to overfit (say = 2000)

Train and Runtime Complexity.

we have seen that we can run different sets of models parallely in RF.

But in Gradient Boosting we can't parallelize it, because, it works serially. ↳



GBDT takes more time to train than RF.

RF :- ↓ $O(n \lg n d * M)$

But Run time Complexity.

↓

GBDT → $O(\text{depth} * M)$ (small)

So, low latency algorithm.

Space time ← $O(\text{depth} * M + \gamma_m)$

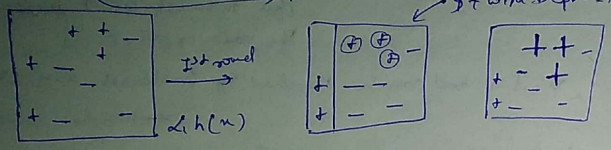
$O(\text{store each tree} + \gamma_m)$

most of the internet companies like, google, use GBDT for low run time complexity.

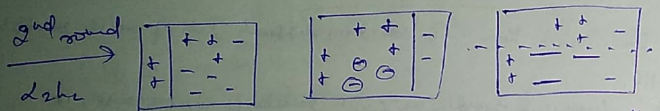
Adaboost Geometric Intuition.

→ mostly used in face detection.

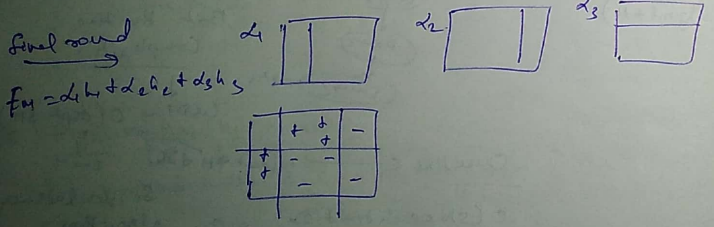
→ DT with Depth = 1



Working of adaboost. At first it will seperate 2 positive pts. So, it misclassifies 3 true pts. So, next time it increases the weights of 3 true pts.



In this round it correctly seperated all three weighted true values. but misclassifies 3 -ve values. So, this time weight of -ve value will increase.

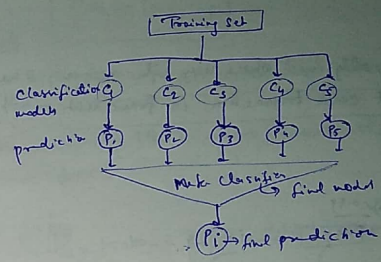


$F_M = \alpha_1 h_1 + \alpha_2 h_2 + \alpha_3 h_3$

Stacking Models

In boosting we build models based on errors, but here in stacking different models are trained independently (parallely).

more different the model, more better is stacking.



C_1 → Log sum
C_2 → RBF sum
C_3 → CART (tree)
:

Suppose we have to train a model, and classify x_q.

Then, x_q will be passed to all different parallel models, then the predicted value from (P_1 to P_{10}) is passed to meta classifier then final prediction is made.

Algorithm :-

Input :- training data $D = \{x_i, y_i\}_{i=1}^{n}$ ($x_i \in R^n$, $y_i \in Y$)

Output :- An ensemble classifier H.

1. Step 1 :- learn first level classifier
2. for i ← 1 to T do
3. learn a base classifier h_i based on D
4. end for
5. Step 2 :- Construct a new datasets from D
6. for i ← 1 to n do
7. Construct a new data set that contains $\{x_i', y_i\}$, where $x_i' = \{h_1(x_i), h_2(x_i), \ldots h_T(x_i)\}$
8. end for
9. Step 3 :- learn a second-level classifier
10. learn a new classifier h' based on the newly constructed dataset.
11. return $H(x) = h'(h_1(x), h_2(x) \ldots h_T(x))$

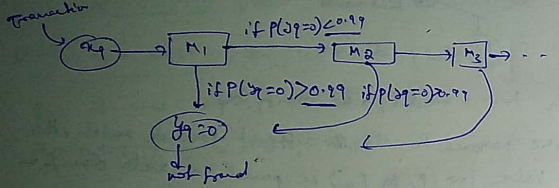
pip2 install mlxtend

→ for stacking.

Cascading Models

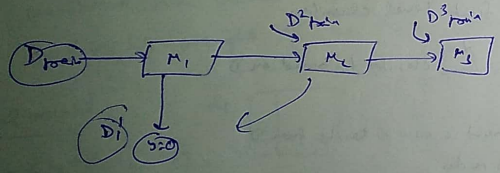
↳ Cascading models is generally used where very high accuracy is needed. like in credit card transaction is fraudulent or not.

Suppose { 0 → not fraud
1 → fraud }



When the model fail to clarify then a final call is made to customer about the transaction.

↳ Cascading is used when the cost of making a mistake is high.
Eg:- Cancer or not Cancer.



$D_{train} \rightarrow M_1$,
$D^2_{train} \rightarrow M_2$
$D^2_{train} = D_{train} - D'$
$D^3_{train} = D^2_{train} - D''$

Deep learning

14:11

History of Networks and Deep learning.

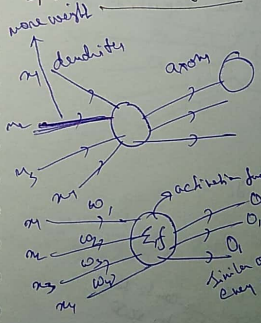
Perceptron → 1957 → by [Rosenblatt]
discovered in

In 1986 → Hinton & others added Backpropagation algo to NN
↳ this is - Simple chain rule.

After 1986 → there was a lot of hype about AI. But due to less Computation power and data, NN was not so much popular. but SVM, logistic regr was.

NN gets its popularity in 2012 when it perform supremely well in ImageNet Competition.

How biological Neuron work



{ activated fired }
↓
when in neuron, there is enough data as input then it fires.
data with more weight are more important.

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

$O = f\left(\sum_{i=1}^{n} w_i x_i\right)$
output, activation, weights, inputs

Growth of Biological neural networks.

↳ when a child gets birth it has less interconnection in brain, but as he grows up from 0 to 6 year old. It learn most of the thing, like, walking, seeing different things etc. As he grows older, again, interconnection of NN or brain decreases.

The neuron which is thicker has higher weight.

Learning happen when, neuron connects with edges.

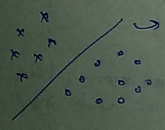
Since, a japanese student learns japanese and an indian learns Hindi, so, this also creates different weights in neuron.

As we are studying and writing right now, it also creates some weights.

Diagrammatic Representation logistic regression and Perceptron.

simple NN model.

↳ We have seen LR earlier, it creates a linear separation.



↳ geometric

LR :- $x_i \longrightarrow \hat{y}_i \longrightarrow$ 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 → w, b ($w \in R^d$, $b \in R$) can be found by SGD

In logistic regression sigmoid acts as an Activation function.

But in perceptron if $w^T x_i + b > 0 \longrightarrow 1$

$w^T x_i + b < 0 \longrightarrow 0$ ↳ activation function

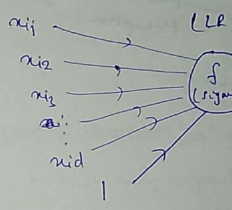
In perceptron, if $w^T x_i + b > 0 \rightarrow 1$ neural network fires, otherwise not.

In LR ↳ $\hat{y}_i = \text{sigmoid}\left(\sum_{j=1}^{d} w_j x_{ij} + b\right)$

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

$w = [w_1, w_2, w_3, \ldots, w_d]$

In perceptron. $O = f\left(\sum_{j=1}^{d} w_j x_{ij}\right)$

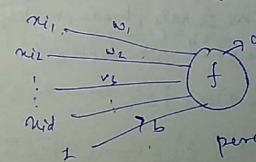


$\hat{y}_i = f(w_1 x_{i1} + w_2 x_{i2} + \ldots w_d x_{id} + 1 + b)$

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

So, LR can be represented as neural network.

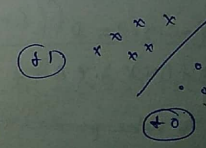
So, train a NN, weights on edges/vertices.



activation fn $\Rightarrow f(x) = \begin{cases} 0 & \text{otherwise} \\ 1 & \text{if } w^T x + b > 0 \end{cases}$

↳ perceptron

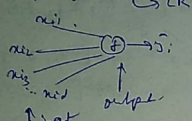
↳ perceptron is simply a linear classifier.

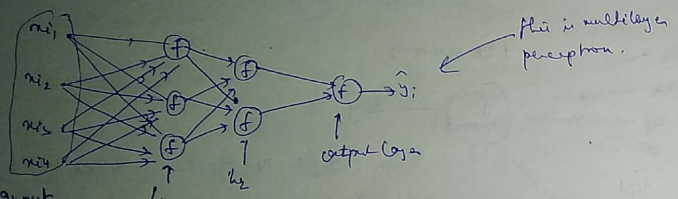


But in LR Separation is done by sigmoid

Multi-Layered Perceptron (MLP)

Till now we saw Single layer perceptron ↳ LR





Q Why should we care about MLP ?

↳ because it is ~~~~ inspired from neuroscience, brains of rats, human, monkeys.

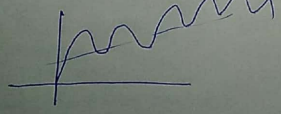
in which each layer is multi connected with different layer.

Mathematical representation $\{x_i, y_i\} = D$

↳ $y = f(x)$ $y \in R$

Suppose we have to make prediction on $f(2 * \sin(x^2) + \text{sqrt}(x * 5))$

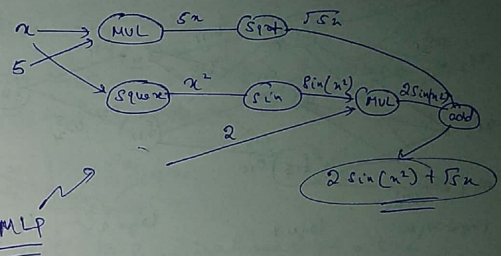
So, if we do LR, or Single perceptron then, this model will make a lot of errors,



But, this model can be solved easily by multilayer perceptron.

Sol: $F(x) = 2\sin(x^2) + \text{sqrt}(5x)$

$f_1 \rightarrow$ add ( )
$f_2 \rightarrow$ sqr ( )
$f_3 \rightarrow$ sqrt ( )
$f_4 \rightarrow$ sin ( )
$f_5 \rightarrow$ Multi ( )



MLP

~~MLP is basically~~ when we need to ~~~~ make prediction on non linearly separable data then MLP can be easily used, but this can easily overfit.

MLP is basically a composite function, $f \circ g(x)$ $g \circ f(x)$

$F(x) = 2\sin(x^2) + \sqrt{5x}$

$\sqrt{5x} \rightarrow f_3(f_5(x,5))$

$2\sin(x^2) \rightarrow f_5(2, f_4(f_2(x)))$

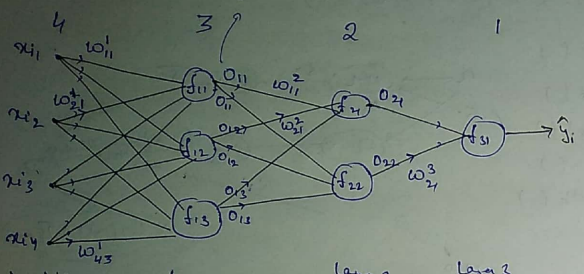
$\therefore F(x) = f_1(f_5(2, f_4(f_2(x))), f_3(f_5(x,5)))$

⇒ By using multi-layered structure we can come up with complex math formulas to solve your problem.

MLP → graphical ways to represent $f \circ g$, $g \circ f$

↳ powerful model

Notation



f_{ij} → layer & index

o_{ij} → layer & index.

w^k_{ij} → k → next layer; from → to

This is a fully connected layer

Since, from 1st to 2nd layer, there is total $4 \times 3 = 12$ connection

Similarly, for 2nd to 3rd → $3 \times 2 = 6$ connection.

So, these weights can be represented as matrix.

$$W^1 = \begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \\ w^1_{31} & w^1_{32} & w^1_{33} \\ w^1_{41} & w^1_{42} & w^1_{43} \end{bmatrix}_{4 \times 3}$$

Similarly, W^2 can be made.

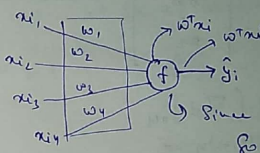
Training a Single-neuron model

→ Here with training we find the best edge weight using D_{Tr}.

perceptron & logistic regr^n → Single Neuron models for classfn.

linear regr^n → Single Neuron model for Regr^n.

As we know in linear regr^n → function f is identity function because input given is same as output.



→ Since input and output is same so, f is identity function.

Linear regr^n

$$\hat{y}_i = \sum_{j=1}^{d} w_j x_{ij} \quad \Big\} \rightarrow \text{linear optimization.}$$
$$\hat{y}_i = w^T x_i$$

$x_i \in R^d$, $y_i \in R$

$$\left\{ \min_{w_i} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \text{regularization} \right.$$

optimization

$$\left\{ \min_{w_i} \sum_{i=1}^{n} (y_i - w^T x_i)^2 + \|w\|_2^2 \right.$$

Steps to train

① Define a loss-function.

$L = \sum_{i=1}^{n} L_i$ → $L = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \text{reg}$ → for all training data

$L_i = (y_i - \hat{y}_i)^2$ → Loss of single train data

② optimization.

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

neglecting regularization for simplicity.

$y_i = f(w^T x_i)$

→ Identity for linear regr^n
Sigmoid for Log. regr^n

$$w^* = \arg\min_w \sum_{i=1}^{n} (y_i - f(w^T x_i))^2 + \text{reg}.$$

(3) Solve the optimization problem.

(a) initialize weights w_i's → random

(b) Compute $\nabla_w L = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \\ \vdots \\ \frac{\partial L}{\partial w_d} \end{bmatrix}$ $w \in R^d$, $x_i \in R^d$

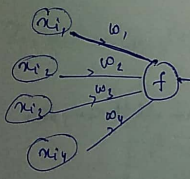
(c) $w_{new} = w_{old} - \eta [\nabla_w L]_{old}$ ← update weight; ↳ learning rate

$$(w_i)_{new} = (w_i)_{old} - \eta \left[\frac{\partial L}{\partial w_i}\right]_{(w_i)_{old}}$$

for itr = 1 to k

Gradient Descent $\nabla_w L$ → x_i's & y_i's → all

SGD → $\nabla_w L$ = for small batch of points SGD.



→ follow path,

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

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial f} \frac{\partial f}{\partial w_i}$$ → Chain rule of differentiation.

we know, $L = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \cancel{\text{reg}}$.

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

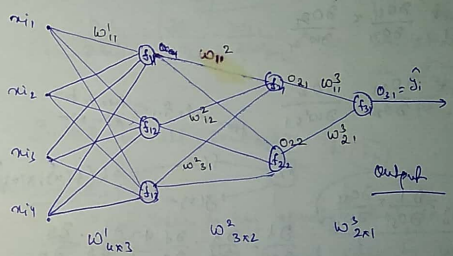
$$\therefore \frac{\partial L}{\partial f} = -\sum_{i=1}^{n} 2(y_i - f(w^T x_i)) \quad \left| \frac{\partial f}{\partial w_i} = x_i \right.$$ ($f = w^T x_i$)

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

Training a MLP Chain rule.

In previous video we learned about training a single neuron-model.

Given, $D = \{x_i, y_i\}$



Here, we need to determine

12 + 6 + 2 = 20 weights and update them.

(1) $L = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + (\text{reg}) \rightarrow \sum_{i,j,k} \{w^k_{ij}\}^2$

(y_i ← actual, $\hat{y}_i$ ← predicted; y_i-true)

(2) SGD or GD

(a) initialize w^k_{ij} randomly

(b) update rule: $(w^k_{ij})_{new} = (w^k_{ij})_{old} - \eta \left(\frac{\partial L}{\partial w^k_{ij}}\right)$ ↳ learning rate.

(c) perform updates till convergence

Now, to update weight.

$$\frac{\partial L}{\partial w^3_{11}} = \frac{\partial L}{\partial o_{31}} \times \frac{\partial o_{31}}{\partial w^3_{11}} \leftarrow \text{chain rule}$$

$$\frac{\partial L}{\partial w^3_{21}} = \frac{\partial L}{\partial o_{31}} \times \frac{\partial o_{31}}{\partial w^3_{21}}$$

$$\frac{\partial L}{\partial w^2_{11}} = \frac{\partial L}{\partial o_{31}} \times \frac{\partial o_{31}}{\partial o_{21}} \times \frac{\partial o_{21}}{\partial w^2_{11}}$$

$$\frac{\partial L}{\partial w^2_{21}} = \frac{\partial L}{\partial o_{31}} \times \frac{\partial o_{31}}{\partial o_{21}} \times \frac{\partial o_{21}}{\partial w^2_{21}}$$

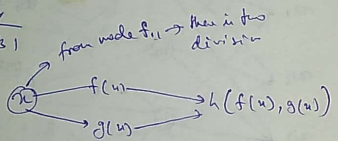
$$\frac{\partial L}{\partial w^2_{31}} = \frac{\partial L}{\partial o_{31}} \times \frac{\partial o_{31}}{\partial o_{21}} \times \frac{\partial o_{21}}{\partial w^2_{31}}$$

For update of w^1_{11},

$$\frac{\partial L}{\partial w^1_{11}} = \frac{\partial L}{\partial o_{31}} \times \frac{\partial o_{31}}{\partial w^1_{11}}$$

~~$\frac{\partial L}{\partial o_{31}} \times \frac{\partial o_{31}}{\partial o_{21}} \times \frac{\partial o_{21}}{\partial w^1_{11}}$~~

from node f_{11} → there is two division



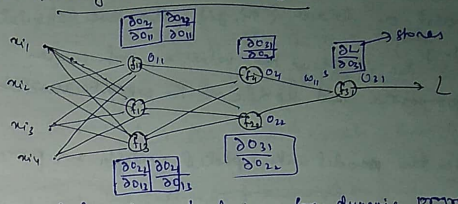
$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial f} \cdot \frac{\partial f}{\partial x} \text{ (+) } \frac{\partial h}{\partial g} \cdot \frac{\partial g}{\partial x}$$

↳ Sum.

$$\frac{\partial o_{31}}{\partial w^1_{11}} = \frac{\partial o_{31}}{\partial o_{21}} \cdot \frac{\partial o_{21}}{\partial o_{11}} \frac{\partial o_{11}}{\partial w^1_{11}} + \frac{\partial o_{31}}{\partial o_{22}} \cdot \frac{\partial o_{22}}{\partial o_{11}} * \frac{\partial o_{11}}{\partial w^1_{11}}$$

$$\therefore \partial w^1_{11} = \frac{\partial L}{\partial o_{31}} \cdot \left\{ \frac{\partial o_{31}}{\partial o_{21}} \cdot \frac{\partial o_{21}}{\partial o_{11}} \cdot \frac{\partial o_{11}}{\partial w^1_{11}} + \frac{\partial o_{31}}{\partial o_{22}} \cdot \frac{\partial o_{22}}{\partial o_{11}} \cdot \frac{\partial o_{11}}{\partial w^1_{11}} \right\}$$

Training an MLP Memoization



Memoization is a technique from dynamic programming.

As we have seen earlier, while apply chain rule,

$$\frac{\partial L}{\partial w^2_{11}} = \left[\frac{\partial L}{\partial o_{31}} \frac{\partial o_{31}}{\partial w^3_{11}}\right]$$

$$\frac{\partial L}{\partial w^3_{21}} = \left[\frac{\partial L}{\partial o_{31}}\right] \cdot \frac{\partial o_{31}}{\partial w^3_{21}}$$

this term has used a lot of time. So, with memoization, we store the value of $\frac{\partial L}{\partial o_{31}}$ in f_{31}. So that we need not to recompute it each time.

Similarly, we will save $\frac{\partial o_{31}}{\partial o_{21}}$ to f_{21}

Memoization

↳ if there is any operation that is used many times repeatedly, it's a good idea to compute it once, store it and reuse it

↳ it boost up the speed, (uses slightly more memory)

For time, we need to store only 7 values, and can use a lot of time.

Chain rule + memoization

↓

Back propagation.

Back Propogation

Firstly we train our model on dataset $\{x_i, y_i\}$. Then we calculate the loss $L(y - \hat{y}_i)$ ↳ if these two values are not similar then we backpropogate and update all the weights associated with network.

We again train the model with updated weights and process continues, till we get $(w_{new} = w_{old})$

Steps

Data ⇒ $D\{x_i, y_i\}$

1. initialize w^k_{ij}'s
2. for each x_i in D
   - (a) pass x_i forward through the network → forward propogation
   - (b) Compute $L(\hat{y}_i, y_i)$
   - (c) Compute all the derivatives chain rule & memoization
   - (d) Update the weights from end of the network to start. → this is backward propogation.

Let say, from previous network.

$$(w^3_{11})_{new} = (w^3_{11})_{old} - \eta\left(\frac{\partial L}{\partial w^3_{11}}\right)$$

So, what ever Activation function we take it should be easily differentiable, because if not then, we can't use chain rule.

3. repeat step 2 till Convergence

Convergence occurs when

$$(w^k_{ij})_{new} \cong (w^k_{ij})_{old}$$

Since, suppose, we send the input data and get the output and updated the weight ↳ this is one epoch.

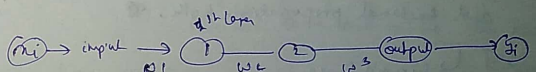
Now, we again send the updated weight forward and this step continues ↳ this is 2nd epoch, then 3rd epoch...

Backprop → chain rule + memoization

1. input
2. epochs → forward prop, Compute loss, Compute derivatives (chain rule & memoization), update weights, backprop

process repeated till convergence

V.V.I ⇒ In backprop activation function must be differentiable. It speeds up the training of the NN.



So, as a input we can use full data at a time also, but this takes a lot of memory. (Gradient Descent)

or we can take each point at a time, this is (SGD)

or we can take batch of points at a time (mini batch SGD) ↳ most popular approach.

Suppose 10 lk points in D

mini batch = 100

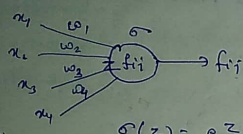
epoch = $\frac{10000}{100} = 100$.

2. for each epoch mini batch of size 100, do, forward prop + loss + $\frac{\partial L}{\partial w_{ij}}$ + update + backprop.

Activation Function.

The Concept of activation function was first introduced in 1990 & 90s.

Generally, Sigmoid & tanh is mostly used.



It should be easily differentiable

z is passed in function f_{ij}

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

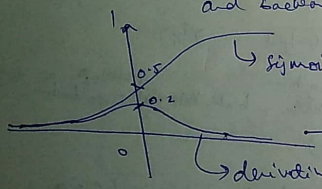
$$w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

$$z = w^T x$$

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

Here we can see that derivative of Sigmoid can be used as a function of Sigmoid. So, it can be used for forward and backward propagation both.

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

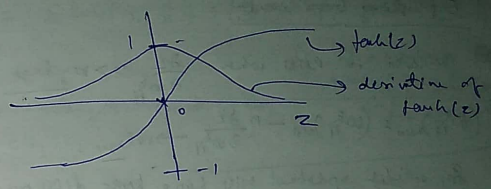


$$0 \le \frac{d\sigma}{dz} < 1$$

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

$$\frac{da}{dz} = 1 - a^2$$

derivative of tanh(z) can also be represented as a func^n of tanh(z) So, easy in Computation.



Vanishing Gradient ~~Descent~~ Problem.

During 80's and 90's problem of Vanishing Gradient ~~Descent~~ arises in NN.

As the activation function is Sigmoid and tanh, so derivative lie in some value b/w 0 to 1.

So, during weight update.

$$\frac{\partial L}{\partial w'_{11}} = \frac{\partial L}{\partial o_{21}} \left[\frac{\partial o_{21}}{\partial o_{11}} \times \frac{\partial o_{11}}{\partial o_{11}} \times \frac{\partial o_{11}}{\partial w'_{11}} + \frac{\partial o_{21}}{\partial o_{12}} \times \frac{\partial o_{12}}{\partial o_{11}} \times \frac{\partial o_{11}}{\partial w'_{11}} \right]$$

$0 < \cdot < 1$ $\quad$ < 1 $\quad$ < 1 $\quad$ < 1 $\quad$ < 1 $\quad$ < 1

↓

Hence will come out to be very less, nearly zero.

So, $w_{new} \simeq w_{old}$

↳ So, weight are not updated when size of hidden layer are large.

$$(w'_{11})_{new} = (w'_{11})_{old} - \eta \frac{dL}{dw'_{11}}$$

e.g. $= [0.2 \times 0.1 \times 0.05]$
$= 0.0010$
$\ll 1$.

So, $\frac{\partial L}{\partial w_{ij}} \to$ v. v. small because of chain rule.

Exploding Gradient

It occurs in ANN when $\frac{\partial L}{\partial w^k_{ij}}$ >> very large.

$$(w^k_{ij})_{new} = (w^k_{ij})_{old} - \eta \frac{\partial L}{\partial w^k_{ij}} \rightarrow \text{very large}$$

So, weight updated will have huge difference from previous weight.

↳ It also occurs due to chain rule.

$$\frac{\partial L}{\partial w_{11}} = \frac{\partial o_{31}}{\partial o_{21}} \times \frac{\partial o_{21}}{\partial o_{11}} \times \frac{\partial o_{11}}{\partial w_{11}} >> \text{large}$$

(each factor > 1)

Bias-Variance Tradeoff :-

In MLP, there is high chance of overfitting.

So, regularization is added.

Hyperparameter tuning is done in

↳ λ and layers.

① when layer ↑ ⇒ more weights/parameter

So, higher chance of overfitting.

‖

high Variance.

② When layer ↓ (λ ↓) ⇒ higher chance of underfitting.

high bias.

(↳ simple model)

So, in MLP → with multiple layers

↳ for optimization → regularization is added

we → $L = \sum_{i=1}^{n} loss(y_i, \hat{y}_i) + \sum_{i,j,k} (w^k_{ij})^2$

↳ It is similar to [illegible] either L_1 regularization or L_2 can be used.

And L_1 regularization creates a lot of sparsity.

So, $L = \sum_{i=1}^{n} loss_i + \lambda L_2$ reg. on weights

↓

layer λ → lesser weights

λ ↑ → variance ↓

layer ↑ ⇒ Variance ↑.

Deep multilayer perceptrons 1980s to 2010s

from 1980s to 2008 to 2010:
- → people were using 2-3 layered networks.
- → problem was vanishing gradient
  - → too little data → high chances of overfit
  - → too little computing → taking too much time

So, due to these problems, training a deep MLP was tough, Since, it was impossible to handle 1000's of weights and then to update it.

So, after 2010:
- → we get a lot of data → by Google, Facebook etc (labelled or unlabelled)
- → Computing power increased
  - → GPU → this worked really faster for deep learning nearly 100 times faster than CPU.
  - → CPU
- → a lot of new ideas and algorithms developed.

Modern DL: Classical ML → was based on theory and then experiment

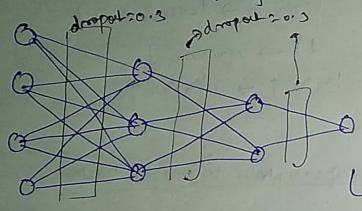
But Modern DL is based on → first experiment then theory.

Dropout & Regularization.

Deep NN → easily overfit ←
↳ many layers → many weights

So, Regularization (l_1 & l_2) can be added to reduce overfitting.

But in 2012 Nitish Srivastava, gave the idea of Dropout which also reduces overfitting in a much more elegant way.



(Let's say dropout = 0.3 means, reduce 30% of weight)

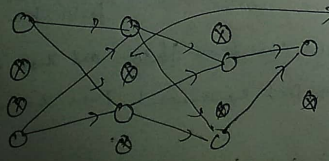
So, the idea of dropout is as follow:

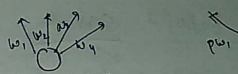
(i) randomly reduce the no. of weights (layer) in each iteration.
↳ then train the model and backpropgate.
↳ then again randomly reduce the no. of connected weights and the process is repeated.

→ This way we are not training on whole data, but a different sets of reduced weights (layer), so, reduces, overfitting, like, we train a model in RF on different set.

(ii) → then during testing we multiply each weight to 0.3 because it was reduced to 30% during training.

This way dropout works.





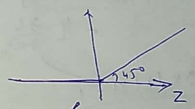
$p = 0.5$ ← dropout

$0 \leq p \leq 1$
dropout ↳ hyperparameter

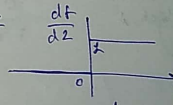
Rectified Linear Units (ReLU)

Since, with sigmoid and tanh function, we had a problem of Gradient Descent.

So, in 2012 ReLU was developed, best activation function till now.



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

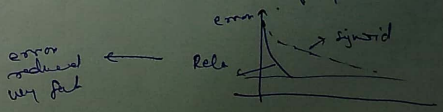


↳ this derivative is either 0 or 1.
So, there is no vanishing gradient, or gradient boosting condition with Relu.
↳ when $z=0$, derivation is 0
this is called the dead activation.

but since it is not differentiable at $z=0$.

So, a similar type of function is taken as Relu

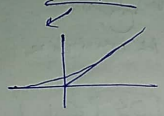
In experiment people have seen that with ReLU as activation function, error reduces with less no. of epochs



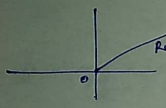
A smooth approximation of ReLU is Softplus.

$f(x) = \log(1+\exp x)$

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



Since, derivative of ReLU involves only 0 and 1 so, it is faster, unlike Sigmoid and Tanh.



When z is -ve $f(z) = 0$

$\frac{df}{dz} = 0$

So, in weight update $(w^k_{ij})_{new} = (w^k_{ij})_{old} - \eta \frac{\partial L}{\partial w^k_{ij}}$

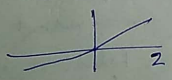
one of the ~~weight~~ derivative is 0, then the whole value becomes 0, which is not good.

$= 0 \times 1 \times 1 \times 1$

$= 0$

So, to overcome this condition we can use Leaky ReLU

$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$



↳ Here instead of value 0, where have $a = 0.01$ which is not 0, but here, we can face problem of ~~gr~~ vanishing gradient.

Potential problem with ReLU

→ Non differentiable at 0

→ Non-zero centered

↔ Unbounded ↳ but still better than ~~ReLU~~ Sigmoid and Tanh.

Weight Initialization.

As we have seen in logistic regression ↳ initialization of weight was random ↙ uniform ↘ random

↳ $w_{ij} \sim N(0, \sigma)$

During initialization of weight of MLP we should never do following steps.

① initialize $w^k_{ij} = 0$ or any const value ✗ $\forall i,j,k$ → V.V. bad.

↑ good

symmetry { → because all the neuron compute the same thing and → same gradient update.

② initialize w^k_{ij} = large -ve number

because, for -ve z in $f(z) = 0$ ReLU.

$W^T x = z$ ← large -ve value

so if W = -ve, z = -ve

x is generally normalized ∴ b/w 0 and 1

mean centering, variance scaling

* So data normalization is mandatory.

So the idea is

① weights should be small (not too small)

↳ not all zero or constant value

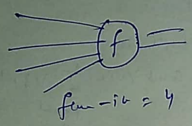
↳ good variance

So, $w^k_{ij} \sim N(0, \sigma)$ ← σ small

↳ so weight should be normalized with 0 mean and constant variance σ^2.

gaussian or normal distributed.

→ So, with lots of research / experiments scientists have come up with lots of different initialization technique, using fan-in and fan-out (but there is no concrete agreement amongst all researchers)



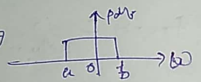
fan-in = 4 , fan-out = 2.

② Uniform - initialization

$$w_{ij}^{k} \sim \text{Uniform}\left[\frac{-1}{\sqrt{\text{fan-in}}}, \frac{1}{\sqrt{\text{fan-out}}}\right]$$

(a = $\frac{-1}{\sqrt{\text{fan-in}}}$, b = $\frac{1}{\sqrt{\text{fan-out}}}$)

weights can be taken any value b/w a and b randomly. →



③ Xavier/Glorot initialization (2010) → better for sigmoid

ⓐ Normal $w_{ij}^{k} \sim N(0, \sigma_{ij})$ $\sigma_{ij} = \frac{2}{\text{fanin} + \text{fanout}}$

ⓑ Uniform $w_{ij}^{k} \sim U\left[\frac{-\sqrt{6}}{\sqrt{\text{fanin} + \text{fan-out}}}, \frac{\sqrt{6}}{\sqrt{\text{fanin} + \text{fan-out}}}\right]$

④ He - initialization (2015) for Relu

ⓐ Normal $w_{ij}^{k} \sim N(0, \sigma)$ $\sigma = \sqrt{\frac{2}{\text{fanin}}}$

ⓑ Uniform $w_{ij}^{k} \sim U\left[-\sqrt{\frac{6}{\text{fanin}}}, +\sqrt{\frac{6}{\text{fanin}}}\right]$

Batch Normalization.

Suppose we have a fully connected layer.

$x \rightarrow L_1 \rightarrow L_2 \rightarrow L_3 \rightarrow L_4 \rightarrow L_5 \rightarrow \text{output} \rightarrow \hat{y}$

input   layer 1

fully connected layer.

Since, x is generally, preprocessed
↳ mean centering
↳ variance scaling.

$D = \{x_i, y_i\}$

$\left\{ \hat{x}_i = \frac{x_i - \mu}{\sigma} \right\}$

Since we do mini batch SGD.

So, for every batch there, is always some change in data.

So a small change in batch input can lead to large change in deep NN layers.

Since, at each stage batch input changes to small small fraction.

∴ if

1	1.01 ← changed value	
×	×	
1	1.01 = 1.0201 × 1.01	
×		
1 = 1	= 1.0406	
⋮		
1	1.325	↪ So till it reaches to layer 5 it has changed to 32 %.

(const value)

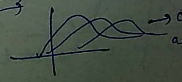
So, this can cost us a lot (known as internal covariance shift)

So, we do ~~batch~~ normalization, before every layer to get the same normalized value, to train much better.

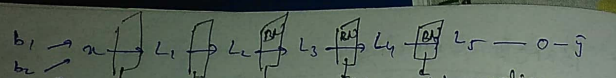
$x - L_1 - L_2 - L_3 - L_4 - L_5 - O - \hat{y}$

{ } = batch

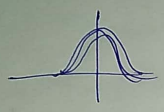
batch 1
batch 2
⋮



→ changes in distribution at stage 5.



Note: for every batch each layer will encounter similar distribution.



Algo:- Value of x over a mini-batch: $B = x_{1,\dots,m}$.

Input → parameters to be learned: γ, β

Output → $\frac{1}{m}\sum_{i=1}^{m} x_i \rightarrow \mu_B$ mini batch mean

$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$ mini batch Variance

$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$ normalize

ε → some error is added to escape ∞.

$y_i \leftarrow \gamma \hat{x}_i + \beta = BN_{\gamma,\beta}(x_i)$

scale and shift

So, with batch normalization:

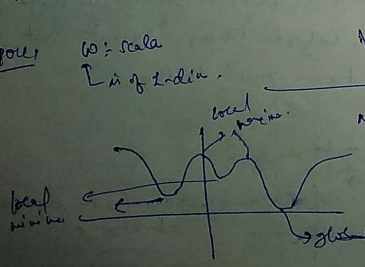
1. Convergence is faster
2. Reduces internal Covariance shift.
3. BN + dropout

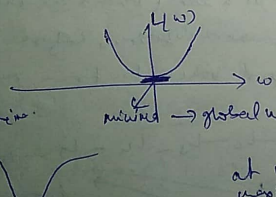
Optimizer Hill-descent analogy in 2D

With optimization we mean to find the optimal value of w^* (the place) for which gives minimum loss. (in LR, linear Regr^n)

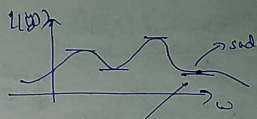
Suppose w = scala

L is of 2-dim.





at minima and maxima $\frac{dL}{dw} = 0$



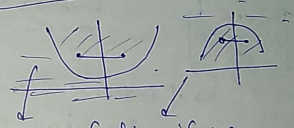
at maxima and minima $\frac{dL}{dw} = 0$

and at saddle point also $\frac{dL}{dw} = 0$

Although slope is 0. but finding maxima and minima is difficult.

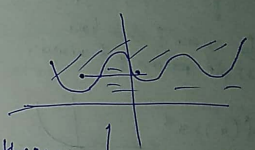
for, at saddle point situation becomes a bit different for algo to perform well.

Convex function & Non Convex function



At Convex function, if we take any two point and join them, it will be inside the function

At Convex function, there is only one maxima or minima



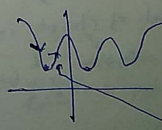
→ at non-convex function, line joining two points may lie outside of it.

At non-Convex function, there is lot of maxima and minima.

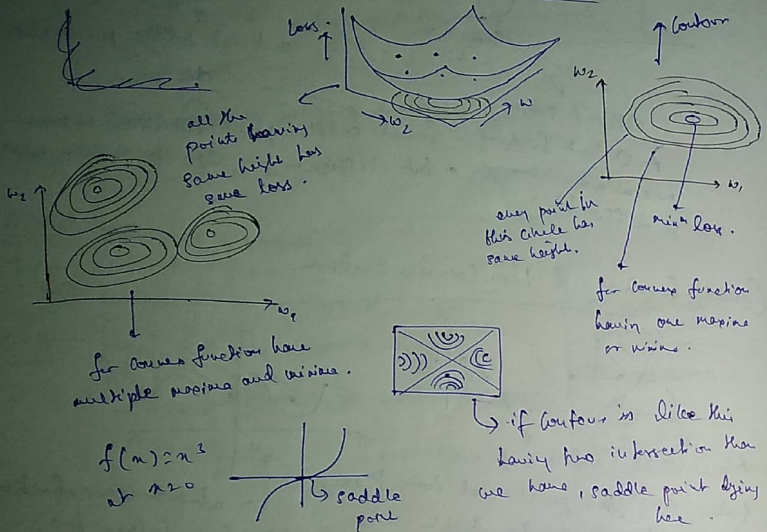
* In logistic reg^n, regr^n and svm → there is only one local minima = global minima

→ Loss function → Convex in nature

But in DL → loss function is non-Convex because there is multiple, minima and maxima.



→ It all depends on where we have initialized the weight. if it around local minima then after reaching loc. minima, it becomes too difficult to climb up.



SGD Recap

Update $(w^*_{ij})_{new} = (w^*_{ij})_{old} - \eta \left[\frac{\partial L}{\partial w^*_{ij}}\right]_{(w^*_{ij})_{old}}$

weight update function.

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

for all $D = \{x_i, y_i\}_{i=1}^{n}$

update $\frac{\partial L}{\partial w}$ → using all the n points in D → GD

↳ using only one pt x_i @ random → SGD

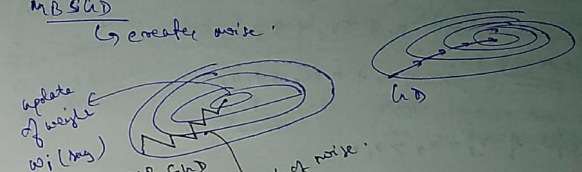
↳ using a random subset of (say 100) pts in D → mini batch SGD

mini batch GD doesn't give exact value as GD, it gives approximate values.

$\left(\frac{\partial L}{\partial w}\right)_{MB\text{-}SGD} \overset{approx}{\sim} \left(\frac{\partial L}{\partial w}\right)_{GD}$

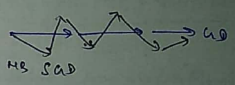
MB SGD

↳ creates noise.



↳ MB SGD is estimation of gradient descent with SGD

↳ each of the updates are more noisy in SGD than G.D



So, for the convergence of MB SGD, denoising of gradient is done

Batch SGD with momentum.

Suppose, these are random numbers.

a_1 a_2 a_3 a_4 a_5 a_6 ...

time = 1 $t=2$ $t=3$ ...

So, at time t_1 → say $v_1 = a_1$

$t=2 \Rightarrow v_2 = \gamma v_1 + a_2$ where, $0 < \gamma < 1$ ← gamma.

So, when $\gamma = 1 \rightarrow v_2 = v_1 + a_2 = a_1 + a_2$

$\gamma = 0.5 \rightarrow v_2 = 0.5 v_1 + a_2$

$= 0.5 a_1 + a_2$

$\gamma = 0 \rightarrow v_2 = a_2$

Here, we can see that new output v_2 is always dependent on previous output a_1, but (a2) has more weight than (a1).

$\Rightarrow v_1 = a_1$

$v_2 = \gamma v_1 + a_2$

$v_3 = \gamma v_2 + a_3 = \gamma(\gamma v_1 + a_2) + a_3$

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

Let $\gamma = 0.5$ $= 0.5 a_1 + 0.5 a_2 + 1.0 a_3$.

$t=1$ $t=2$ $t=3$.

weight of $a_3 > a_2 > a_1$ ...

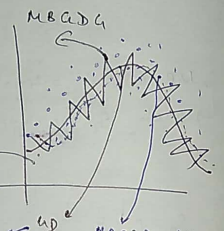
So, doing this in weight update rule reduces in MBSGD.

$v_1 = a_1$

$v_t = \gamma v_{t-1} + a_t$ → v_t = denoised estimate.

$1 > \gamma > \gamma^2 > \gamma^3 \ldots$ denoising of data

this is also known as exponential weighing.



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

↳ Let g_t → g_t

$w_t = w_{t-1} - \eta g_t$

exponential weighing ↳

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

$w_t = w_{t-1} - v_t$

initially at $\gamma = 0$ $v_1 = \eta g_1$

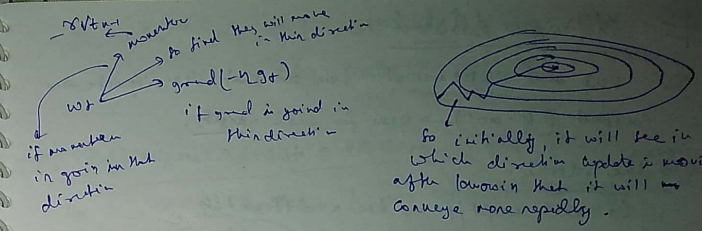
Generally γ is taken as 0.9. $\gamma = 0.9$ → $w_t = w_{t-1} - (0.9 v_{t-1} + \eta g_t)$

So, $w_t = w_{t-1} - [\gamma v_{t-1} + \eta g_t]$

this is called momentum because it depends on previous value $g_{t-1}, g_{t-2}, g_{t-3}, \ldots$

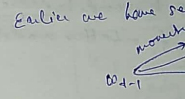
momentum gradient

$\therefore v_t = 1(\eta g_t) + \gamma(\eta g_{t-1}) + \gamma^2(\eta g_{t-2}) \ldots$



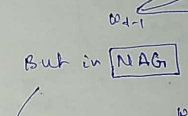
Nesterov Accelerated Gradient (NAG)

Earlier we have seen, SGD + momentum,



→ where final direction travel is the vector sum of gradient + momentum.

But in NAG



→ $w_t = w_{t-1} - (\gamma v_{t-1} + \eta g_t)$

↳ we first travel in the direction of momentum (from weight (w_{t-1})) then we travel in the direction of gradient { point → $(w_{t-1} - \gamma v_{t-1})$

↳ so, here point has changed from $(w_{t-1}$ to $w_{t-1} - \gamma v_{t-1})$.

NAG gives some better result.

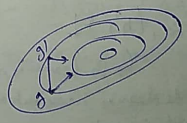
ηg_t → gradient term

γv_{t-1} → momentum

$w_t = w_{t-1} - (\gamma v_{t-1} + \eta g')$

when $g \neq g'$

↳ since starting point has changed.



Optimizers Adagrad

In SGD and SGD + momentum, we see that learning rate was constant ($\eta = 0.01$) → same for each weights.

But here, idea is:

↳ each weight / param has a different η

Because, when we apply BoW, features, become sparse, and for sparse features, constant learning rate is no good.

So, ada grad, η should change such that $\eta \to$ needs to reduce. for a better model.

SGD ↳ $w_t = w_{t-1} - \eta g_t$ → same for all weights

Adagrad ↳ $w_t = w_{t-1} - \eta'_t g_t$ → adaptive.

So, $\eta'_t = \frac{\eta}{\sqrt{\alpha_{t-1} + \epsilon}}$ ($\eta \to 0.01$)

different η' for each weight.

↳ small +ve num (to avoid divisibility by zero)

$\alpha_{t-1} = \sum_{i=1}^{t-1} g_i^2 \to \left(\frac{dL}{dw}\right)^2_{w_{t-1}}$

always +ve. ↳ $g_1^2, g_2^2, g_3^2 \cdots g_{t-1}^2$

So, when time (t) ↑, α_{t-1} ↑, so, (η'_t) ↓

↳ So learning rate is decreasing, automatically which is good.

$\alpha_t \geq \alpha_{t-1}$

So, in Adagrad)

(+) ↳ no need of manually tuning η ↳ $\eta'_t \to$ weight, time / iter

(+) ↳ Sparse features and Dense one can be easily handled

(−) ↳ α_{t-1} can become v. large, as t ↑,

So, when, α_{t-1} ↑↑ with t ↑.

↳ Convergence rate will be slower.

Optimizers Adadelta and RMSProp

↳ idea of adadelta is to use exponential decreasing (decaying) avg. instead of reduce, $\alpha_{t-1} = \sum_{i=1}^{t-1} g_i^2$ ↳ as used in Adagrad.

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

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

exponential decaying avg.

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

Suppose, $\gamma = 0.95$

$eda_{t-1} = 0.95\, eda_{t-2} + 0.05\, g^2_{t-1}$

$eda_{t-1} = (0.05)\, g^2_{t-1} + 0.95\, eda_{t-2}$

$= (0.05)\, g^2_{t-1} + [0.95 \times \{0.05\, g^2_{t-2} + 0.95\, eda_{t-3}\}]$

So, this value is never gonna be too large, so, $\eta'_t \to$ will never be zero.

It has all the features of Adagrad with eda_{t-1} cannot be very large with t ↑.

Adam (Adaptive moment Estimation)

In Adadelta we saw it stores exponential decreasing avg (eda) of $g^2 \to$ learning rate (η'_t)

Idea of adam is it stores eda of g_t also.

It has moment in it because

In statistics, mean called as 1st order moment

variance " 2nd " "

eda ← g_t ← $M_t = \beta_1 M_{t-1} + (1-\beta_1) g_t$ $\quad 0 \le \beta_1 \le 1$

eda ← g_t^2 ← $V_t = \beta_2 V_{t-1} + (1-\beta_2) g_t^2$ $\quad 0 \le \beta_2 \le 1$

Now ↳ $\hat{M}_t = \frac{M_t}{1-(\beta_1)^t}$, $\hat{V}_t = \frac{V_t}{1-(\beta_2)^t}$

$W_t = W_{t-1} - \alpha \frac{\hat{M}_t}{\sqrt{V_t + \varepsilon}}$ $\quad$ if $\beta_1 = 0 \rightarrow$ adadelta.

α usually 1

This is the best optimization technique among all, it converges much faster.

Which Algorithm to choose when.

MB-SGD → momentum & NAG → Adagrad → Adadelta & Rmsprop → adam

- MB-SGD: used for small and shallow DNN
- momentum & NAG: Works well in most cases (but slower)
- Adagrad: good for sparse data
- Adadelta & Rmsprop: faster and best
- adam: fastest and best

Gradient Checking and clipping.

↳ In a normal way we should always keep an eye on gradient on how it is changing. Because, updating gradient is the most important thing in DL.

update weights

So, monitor gradient for each epoch, weights and layers.

because, not doing so, we can face vanishing gradient or exploding gradient.

↳ One solution for exploding gradient is Clipping.

W = all weights in network

G = gradients $\frac{\partial L}{\partial w_{12}}$ $\frac{\partial L}{\partial w_{13}}$ $\frac{\partial L}{\partial w_{14}}$ ... $\frac{\partial L}{\partial w_{1m}}$

let 200 → 100

So, gradient can be so high, so, we clip it by l_2 normalization clipping.

$\tau = 2 \rightarrow$ Cut

When, gradients $\ge \tau$ always

$G_{new} = \frac{G}{\|G\|_2} * \tau$ ↳ threshold

$\|G\|_2 = \sqrt{G_1^2 + G_2^2 + G_3^2 + \dots}$

G: G_1 G_2 G_3 ... G_m

G: 2 | 5 | 10 | ... | 6

$\frac{G}{\|G\|} = \frac{|2|}{<1} \frac{|5|}{<1} \frac{|10|}{<1} \frac{|...|}{<1} \frac{|6|}{<1} * \tau$

If we are vanishing gradient descent, change Sigmoid to Relu.

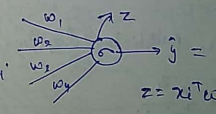
Softmax and Cross-entropy for multi-class classification.

As we have seen logistic regression is used for binary classification, if one want to do multiclass classification, then One Vs Rest method is used.

But a new method is developed using same concept of Logis Reg.

∴ Log. reg. + Multiclass = Softmax.

In Logistic Regn



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

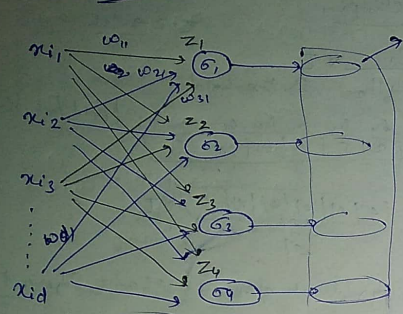
$z = x_i^T w$

↳ So, here, we have only two possibilities.

Softmax classifier. $D = \{x_i, y_i\}$ $y_i \in \{1, 2, 3, \dots K\}$ ↳ multiclass.

$x_i \rightarrow$ [model] → $P(y_i=1|x_i)$, $P(y_i=2|x_i)$, $P(y_i=3|x_i)$, ... $P(y_i=K|x_i)$ → all sum up to 1.

Suppose $k=4$ softmax



Sum to 1

where,

$$\sigma_1(z_1) = \frac{e^{z_1}}{\sum_{i=1}^{k} e^{z_i}}$$

$$\sigma_2(z_2) = \frac{e^{z_2}}{\sum_{i=1}^{k} e^{z_i}}$$

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

Now

$$\sigma(z_1) + \sigma_2(z_2) + \cdots \sigma_k(z_k) = \frac{e^{z_1}}{\sum_{i=1}^{k} e^{z_i}} + \frac{e^{z_2}}{\sum_{i=1}^{k} e^{z_i}} + \cdots + \frac{e^{z_k}}{\sum_{i=1}^{k} e^{z_i}} = \frac{\sum_{i=1}^{k} e^{z_i}}{\sum e^{z_i}} = 1$$

Sum of to 1.

So, Softmax is the generalization of LR to multiclass (each).

Softmax ~~activation~~ minimizes multiclass log-loss

$$\sigma_i(z_i) = \frac{e^{z_i}}{\sum_{i=1}^{k} e^{z_i}}$$

multiclass log-loss $= -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{k} y_{ij} \log P_{ij}$ ← $P(y_i \in i | x_i)$

= cross entropy.

$\{ 1 \text{ if } y_i \in j \; ; \; 0 \}$

Suppose given x_i, $y_i = 3$

1	2	3	4	..	k
0	0	1	0	...	0

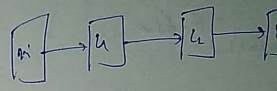
$x_i \rightarrow$ model $\rightarrow$ 0.2 → $P(y_i=1|x_i)$; 0.1 → $P(y_i=2|x_i)$; 0.7 → $P(y_i=3|x_i)$; 0 → $P(y_i=4|x_i)$

$\therefore$ log-loss $= 1 \times \log 0.7$ → one putting on formula.

MLP

2-class

k-class

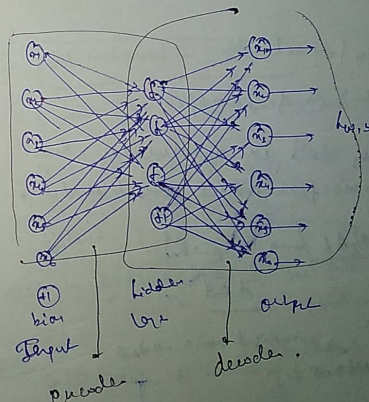


$P(y_i=1|x_i)$
$P(y_i=2|x_i)$
$P(y_i=k|x_i)$

Auto Encoder.

↳ NN which perform dim. reduction like, PCA, t-SNE is
↓ based on variance
↓ Based on Neighbourhood

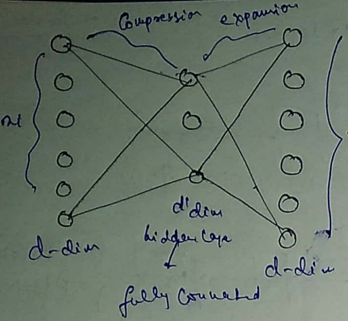
Simple Autoencoder



← Simple autoencoder.

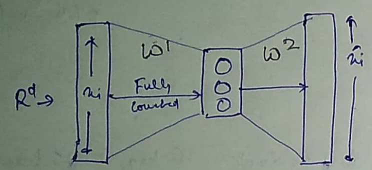
⇒ Auto encoder preserves the information in low dimensional space.

If $x_i \simeq \hat{x}_i$ then, input, output are have preserved more amount of information.



$d' < d$

AE



$L = ||x_i - \hat{x}_i||$

when $x_i \simeq \hat{x}_i$

$L = 0$

Denoising AE

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

we get ↳ $\hat{D} = \{\hat{x}_1, \hat{x}_2, \ldots \hat{x}_n\}$ → noisy.

So, when we pass noisy data in autoencoder, then the output we get is noisefree, more robust model.

$\hat{x}_i \in R^d \rightarrow$ AE $\rightarrow x_i' \in R^{d'}$

$d' < d$

we can add noise to data like

$x_i \rightarrow \hat{x}_i = x_i + N(0, \sigma)$ → normalized noise added

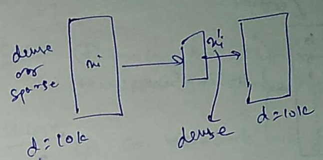
original ↑

↓ AE → $x_i' \in R^{d'}$

Sparse Autoencoder.

to add sparsity, we add L1 regularization to loss.

$|L_1|$ + loss → sparse



* If linear activations are used or only a single sigmoid hidden layer, the optimal solution to an autoencoder is strongly to PCA.

MNIST → 784 dim $\xrightarrow{AE}$ (50 dim) ↳ feature representation

Word2Vec - CBOW

We have seen earlier in Amazon food Reviews → how to convert words to vector using, BOW, TFIDF, W2V. (↓ using gensim)

c_5	c_4	focused	c_1	c_2	c_3
The	Cat	(Sat)	on	the	wall

(The Cat → Context; the wall → Context)

In W2V → CBOW → we take the help of context words to predict focused words.

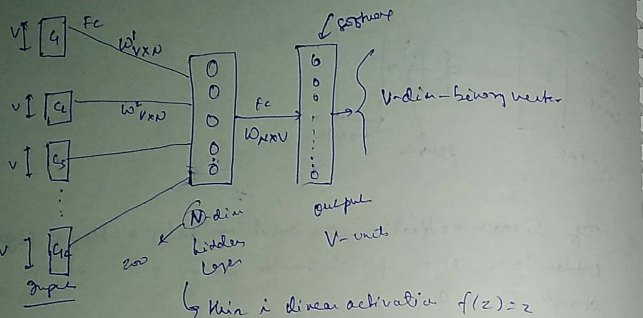
CBOW ↳ ① Create dictionary/vocabulary of words/ ~~vectors~~ = V

V = length of vocab.

② Do one hot encoding of each word.

$W_i \in R^V$: binary vec of V-dim.

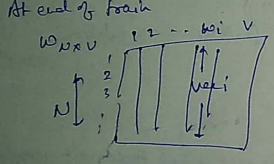
Now, idea is using all the vectors of context words, predict the focused vector using Softmax.



↳ This is linear activation $f(z) = z$

This does not take much time to perform, but Softmax is taken, some time.

At end of train

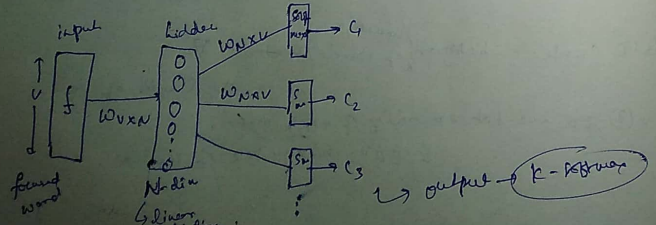


$W2V(w_i) = Vec_i \in N$

Skip-gram: (word2vec)

Earlier we have seen that given context words CBOW W2V-predicts focused word.

But in Skip-gram situation is reversed, here given focused words we have to predict context words, which is quite difficult.



↳ output → K-Softmax

So, No. of weights in CBOW & Skipgram → $(K+1)(N\times V)$

CBOW → 1- Softmax

Skipgram → K Softmax } Skip gram is computationally more expensive.

$K(N\times V) + N\times V$

CBOW
- ⊕ faster than Skipgram
- ⊕ better for frequently occuring words.

Skipgram
- ⊕ Can work with smaller amount of data
- ⊕ Well for infrequent words.

So, in CBOW and Skipgram
we have 'K' no. of context words.
So when K ↑ → N-dim for each point is better

Weights $= (K+1)(N\times V)$

Let $K=5$ $\quad 6\times(200\times 10k)$

$= 1.2$ crore

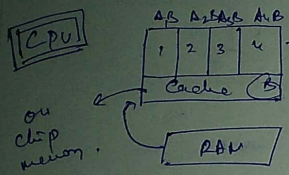
↳ This can take forever to train.

So, optimizer is used to train it more efficiently.

Word2Vec Algorithmic Optimization.

GPU Vs CPU for Deep Learning.

Graphics cards generally developed by NVIDIA for Gaming purpose.



In CPU it is generally of 2, 4, 8 cores, each operates parallely.
like, one for music, one for video, browser.

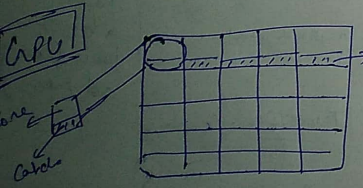
So, Suppose we have to do Computation

$$\begin{bmatrix} \leftarrow A_1 \rightarrow \\ \leftarrow A_2 \rightarrow \\ \leftarrow A_3 \rightarrow \end{bmatrix} \begin{bmatrix} \uparrow \\ B \\ \downarrow \end{bmatrix} = \begin{bmatrix} A_1B \\ A_2B \\ A_3B \end{bmatrix}$$

Here, each matrix multiplication is independent of each other.
So, can operate in parallel.

Cache is used to keep some part of data so that interaction of Core with data becomes easier and fast.

4 parallel:
A_1B | A_5B
A_2B | A_6B
A_3B | A_7B
A_4B | A_8B

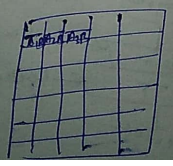


In GPU no. of Cores can be
100
1024
2048
So, each Core operate parallely so, Computation becomes much easier.

each GPU Core Core is much more slower than CPU
→ 400MHz.

Computation

A_1B | $A_{1001}B$
A_2B | $A_{1002}B$
$A_{1000}B$ | $A_{2000}B$



Since here each Core has separate cache & Computation is fast.

16GB/RAM → 3d x 1080 Ti (VRAM video)

49.6 Softmax classifier on MNIST dataset

50-1 Biological Inspiration Visual Cortex

⇒ In this chapter we will learn about (CNN, ConvNets) same
So, CNN is basically used for Visual tasks.
As MNIST data is also a small → Visual task.

In the last topic.
Research on ML/AL + Images/videos → has evolved a lot used for object recognition.

The credit of all this research paper goes to a research paper in 1959 by Hubel & Wiesel (Nobel prize medicine), where he performed an expt. on Cat on how he actually detects images.
While performing the expt he found that a neuron gets fired when it sees some similar pattern.
like →
- → neuron fired
- → neuron little fired
- → neuron little fired
- → neuron doesn't fire
- → so doesn't fired
- →

and our brain has some different layers ($V_1, V_2, V_3 \ldots$)
each layer has different tasks.

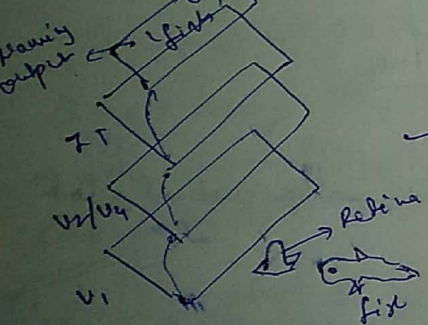
V_1 → detects motion
V_2 → stereo
V_3 → color
V_{3a} → texture segregation
V_7 → face recognition
etc.

Key findings :-

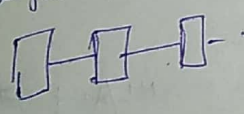
① Some neurons in the visual cortex that fire when it detects presented lines at specific angle.

V_1 ⇒ primary visual cortex → detect edges.

② brain has different layers to detect different things, like, edge detector, motion detector, depth, color detector etc.

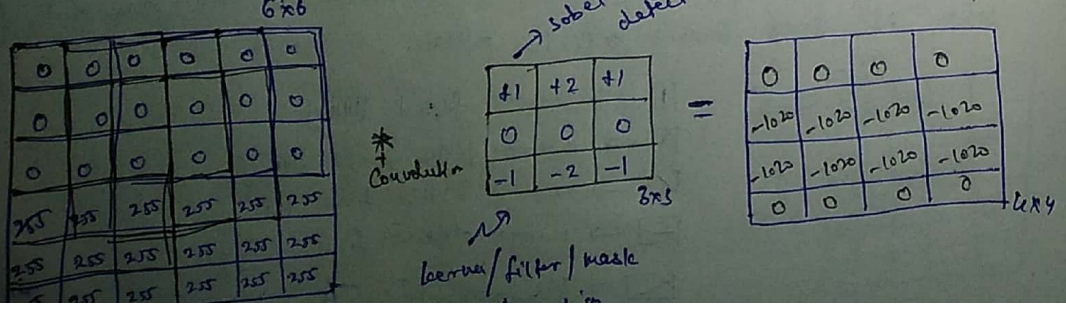


→ each layer operates to detect a fist



Convolution Edge Detection on images

↳ this is the very first layer to detect edges.



$+255 \times 1 + 255 \times (-2) + 255 \times (-1) = -1020$.

↳ we do component wise matrix multiplication & addition and get a 4×4 matrix.

Here, after getting the 4×4 matrix.

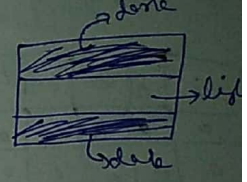
we normalize the value. (since 0 is min and 255 is max.)

for, here, $-1020 \longrightarrow 0$ → min (since -ve)
$0 \longrightarrow 255$ → max

∴ after normalization →

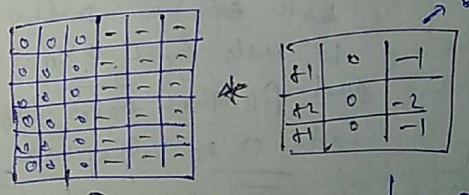
255	255	255	255
0	0	0	0
0	0	0	0
255	255	255	255

image 4×4

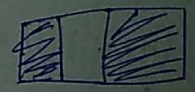


So this is the horizontal edge det

Similarly, horizontal edge detector can be get by following



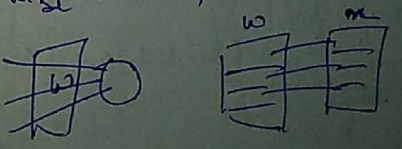
↳ output =



for, convolution is element wise mult; + addition

like MLP → w, u

dot product



Convolution padding and Strides.

As seen earlier, the output we get was of lesser dimension than output.

If input → 6x6 matrix → output = n − k + 1
Sobel edgedetector k → 3x3
= 6 − 3 + 1
= 4

So, now with padding we can achieve output of same dimension as input.

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	0/255
0	255	255	255	255	255	255	0/255
0	0	0	0/255	0/255	0/255	0/255	0/255

8x8

→ 0 when normal padding. / 255 when taking near element

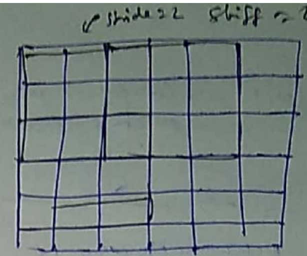
So, here we want output as 6x6 dimension.

So, in n − k + 1 = 6 (k → 3)
∴ n = 8

So we add one extra layer both each sides to make input 8x8.
And fill that extra layer with 0 or value near to it.

So, (n x n) input —(k x k)→ (n − k + 1)(n − k + 1) output
↓
padding.
(n x n) —k x k, padding→ (n − k + 2p − 1)(n − k + 2p − 1)

Strides.
earlier in input we traversed each step.
So, with stride = 2 we traversed 2 step.



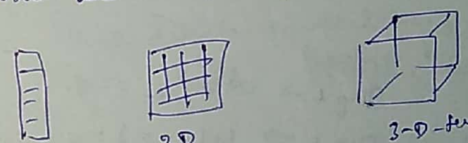
Strides :- (n x n) —S = s, k x k→ $\left(\left\lfloor \frac{n-k}{s} \right\rfloor + 1\right) \times \left(\left\lfloor \frac{n-k}{s} \right\rfloor + 1\right)$

6x6 —s = 2, k = 3→ $\left(\lfloor 3/2 \rfloor + 1\right) \times \left(\lfloor 3/2 \rfloor + 1\right)$
→ we take the floor value of it.
∴ 2x2 (output)

Convolution over RGB images.

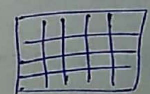
Till now we have seen convolution in grayscale image. (2-D)
Now we will see convolution in 3-D image.

Tensor →

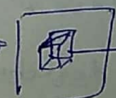


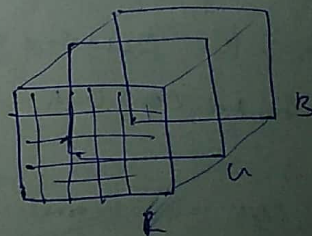
So, colour image is a 3-D tensor.

Colour → grayscale →



→ Color image →



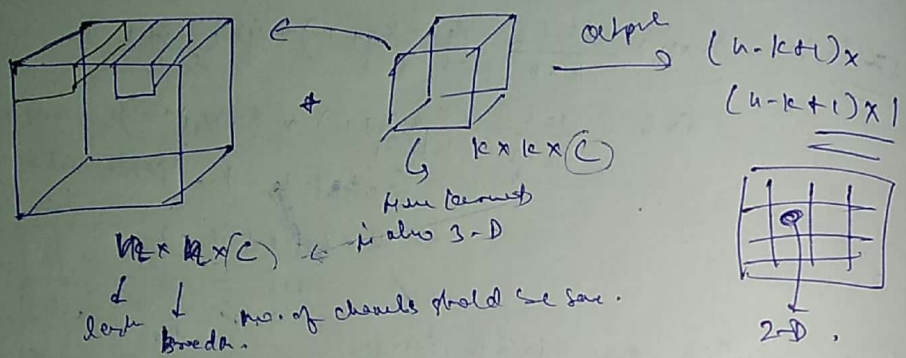


So here each cell has 3 values [255, 0, 155]

So, RGB is referred to as channels → this name comes from signal processing.

As we have seen that
convolution in 2D matrix happens with → Component wise mult + addition

So, similarly,
_______ 3D _______ → _______



Convolution layer.

→ Earlier we saw how a MLP worked

Step 1 → $w^T x = Z$

Step 2 → $ReLU(Z) = y$.

So, convolution layer also works kind of similarly.

(1) Conv. layers are inspired biologically from V_1, V_2, V_3 ...

(2) As in the first stage of seeing, there is multiple edge detector,

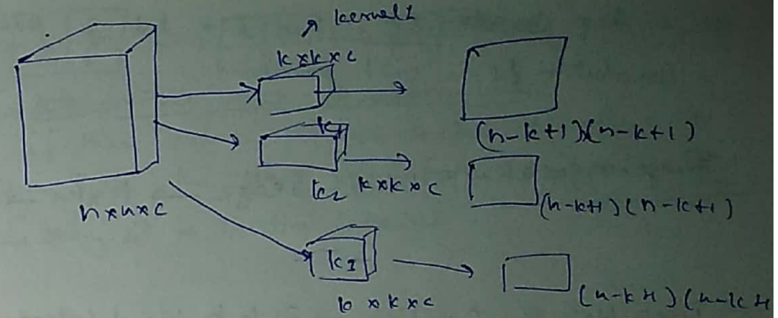
So in Conv. layer. 1-edge detector → 1 kernel
multiple edge detector → multiple kernel.

In CNN also kernels learn from backprop.
In MLP our weights were updating but in CNN, kernels will update.

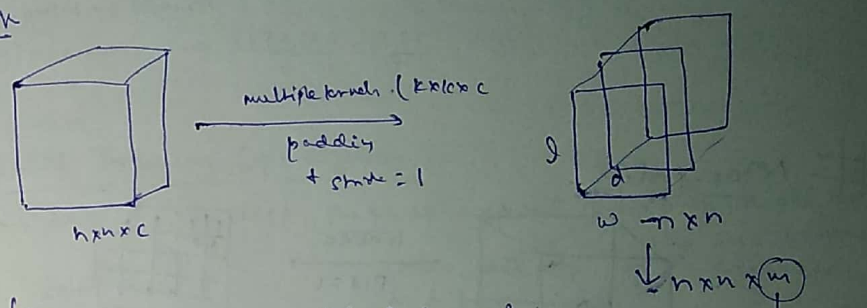
MLP → learn w's → $w \cdot x$ → dot product (weight, input)

CNN :- learn kernel matrices → $I * W_{k \times k}$ → kernel (input, convolution)

$D = \{x_i, y_i\}$



Conv. Layer



So, here, The input image lot of kernels is passed lets say kernel 1 detects horizontal line

k-2 detects parallel horizontal line

k-3 detect 45 degree line

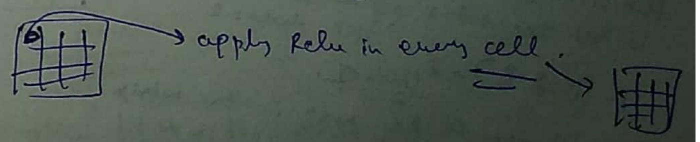
So, corresponding to every kernel, there will be one output. (n x n x 1) = for m kernel (n x n x m)
n x n x1

Here, P, S, k → m hyperparameter.

CNN

Step (1) Input (n x n x c) * kernel 1 (k x k x c) → P, S: 1 → n x n x 1
Convolution

Step (2) → apply Relu in every cell.

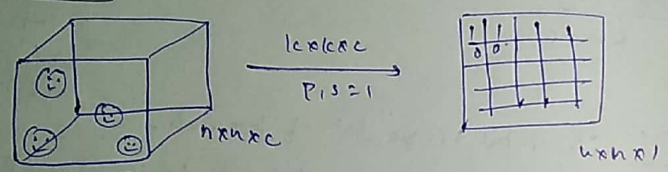


For a deep CNN, there is a lots of pair of Convolution + Relu unit present.

Image recognition :> Pixel → edge → texture → motif → part → object.

Deep Multi Layer ConvNet → Similar to Visual Cortex

Max-pooling.



Inspiration of max^m pooling is taken from location invariant.

this mean, suppose, if has to detect a face then it will detect it, no matter where the image is in the figure, or shape or size of face.

S = 2

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

(max pooling) (k=2) → (S=2)

6	8
3	4

max pooling (k=2), S=1 →

6	7	8
6	7	8
3	3	4

So, when k = (2×2) and stride = 2. this will detect the max^m value in that 2×2 matrix and then traverse with stride = 2

if (k = 2×2) and stride = 1. this will detect the max^m in 2×2 matrix and traverse with stride = 1.

So, Suppose

1	1
5	6

→

1	5
1	6

→

6	5
1	1

↳ ~~rotates~~ the ~~value of matrix~~ within, if these values changes within this matrix, even then the max^m value from max pooling = 6.

that's why it is location invariant.

Two types of pooling.
- ↳ Max-pooling → ConvNets
- ↳ Avg/Mean-pooling → $\frac{1+1+5+6}{4} = 3.25$

CNN Training Optimization.

As we have seen in MLP, that we applied back prop in all the algos like SGD, Adagrad, Adam etc. where back prop is differenciable.

$x \xrightarrow{weight} \hat{y} \rightarrow L(y, \hat{y})$

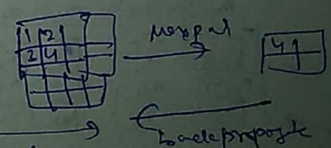
$\frac{\partial L}{\partial w_i}$ → differenciable.

So, here we have to check if Conv. layer and max pooling is differenciable or not.

Since Relu is differenciable so, Conv. layer is also differenciable because in Conv. we do element wise mult + add^n. like in MLP.

MLP ← dotproduct → so differenciable

ConvNet → matrix wise mult + add^n



Now, when talking for Max pool.

So, when it back propagate ~~it~~ ~~gives d~~ derivative is like $\frac{\partial output}{\partial input} = \frac{\partial 4}{\partial 4} = 1$ ↳ same value

and for all other value derivative = 0.

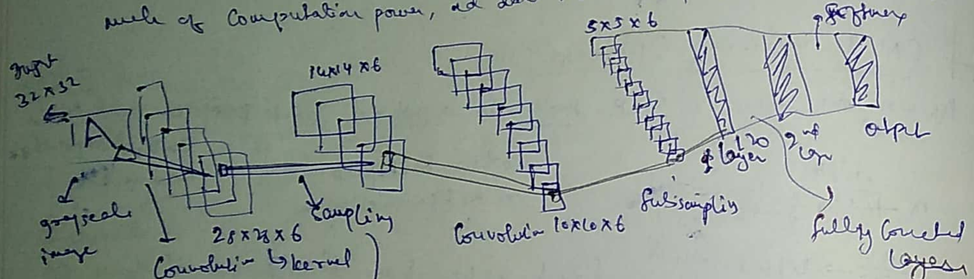
So, max pooling is also differenciable.

$\max(x) \rightarrow \frac{\partial g}{\partial x_i} \begin{cases} 1 & \text{if } x_i = \max \\ 0 & \text{otherwise} \end{cases}$

So, Conv net and max pooling is differentiable so we can apply backprop to it.

Example CNN LeNet [1998]

This was a paper by LeCun in 1998, since we didn't have much of computation power, and data that time, so it was not popular.

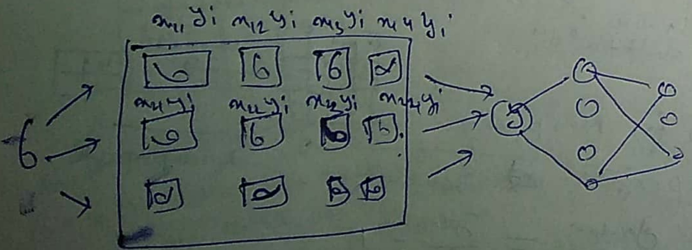


Sampling is mean pooling.

Since that time Adam, Relu, was not developed, so, tanh + sigmoid was used.

Data Augmentation

↳ it means to add extra features to data.



Here each input is augmented, means, for 6 has given different shapes, rotated, shrinked, blurred, bright etc.

So, that for each x_i → there is seven more x_i' → but same y_i.

This way our dataset is increasing and accuracy also increase.

$D = \{x_i, y_i\}$

Why is data augmentation useful.

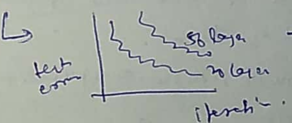
generally we can create lots of invariance in data, like {shift, rotation, zoom, shear, noise} for better result.

→ we can generate large dataset from small one.

→ very popular for image dataset.

Convolution Layer in Keras. [Read paper, github, for provided link.]

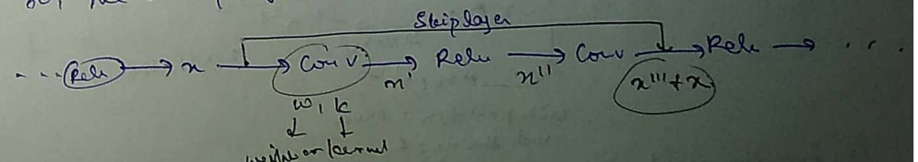
Residual Network.



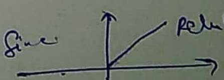
→ In convolution neural network, it is seen that when no. of neural network ↑, training errors and test error also increases.

But this should not happen.

So, the concept of residual network was introduced,



Here since



when x = +ve Relu(x) = x ; Relu(Relu(x)) = x

x = −ve Relu(x) = 0 ; Relu(Relu(x)) = 0.

So when adding some extra layer, if weight or kernel weight of some layer reaches closer to zero, then this layer becomes useless.

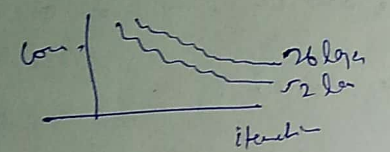
So, with ResNet, we can skip these layers before one relu, the output of next relu is always …

So, this way, our performance increases:

→ if each layer is giving some modification to weight, instead of zero, even then this layer is good.

So, in both situation model is good.

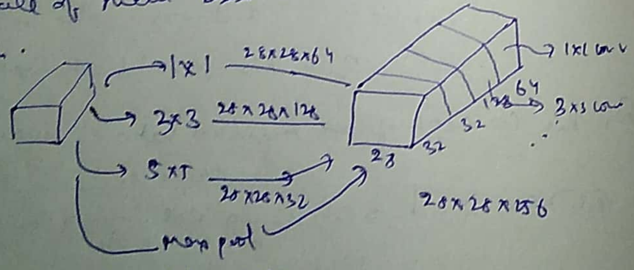
with relu,



| Inception Network |

[paper
↳ Architecture to CNN-inception network.

The idea of inception network is, instead of deciding, whether to use, 1x1 Convolution or 3x3-Conv or 5x5 conv or pooling layer: why not use all of them. But if we use all of them at once, then computation becomes very high.

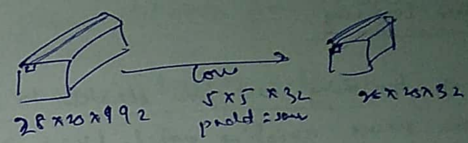


with padding = same and stride = 1.

So, from the input, output generated is stacked on top of each other.

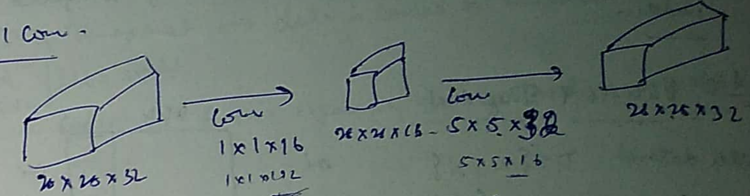
Disadvantage is lot of memory required,

eg:



Total no. of calculation: (28x28x32) * (5*5*192) = 120 million.

Using 1x1 Conv.



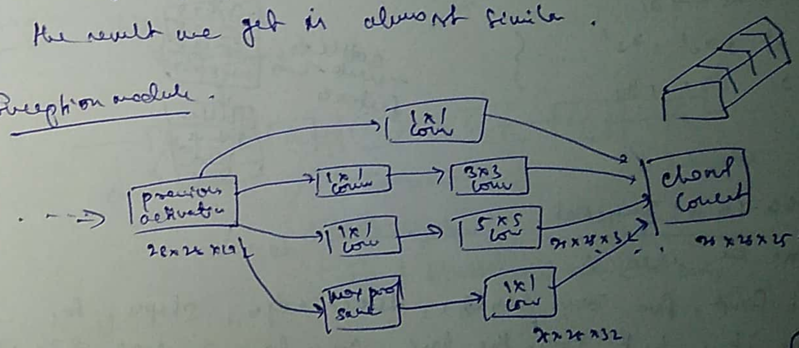
Total Calculation = [(28 * 28 * 16) * (1 x 1 x 192)] + [(28 x 28 x 32) * (5x5x16)]

= 12.4 million

↳ very less than 120 million.

So, before applying 5x5 or more conv. we apply 1x1 conv. and the result we get is almost similar.

Inception module.



So, 1x1 conv is added in every network.

[code in keras

What is Transfer Learning.

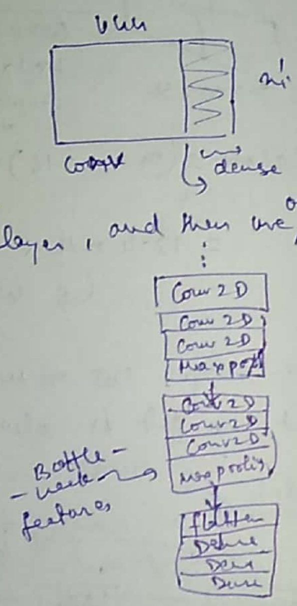
↳ Idea is :- Instead of building NN from scratch to solve on task, we can reuse existing models (VGG16) trained on different dataset.

↳ So here, what VGG-16 has learned is being transfered to different model.

Case 1 :- VGG-16 + ImageNet

Consider dataset $D = \{x_i, y_i\}$
↳ image.

Instead of training in all dataset, use pretrained weights till conv. layer, and then use own model for last fully connected layer.



So for each
$x_i' \to x_i' \to y_i$

$D' = \{ x_1', x_2', x_3' \ldots ; y_1, y_2, y_3 \}$

Case 2 :- VGG-16 + ImageNet

↳ Here the idea is first, first, few conv. layers, detect edges, shapes, so, retain it, but train the last conv. layer + fully connected layer, keeping small learning rate.

Case 3 :- VGG16 + ImageNet ⟵ initial model.

↳ keep this as initial model and tune the complete model on $D = \{x_i, y_i\}$ + keeping learning rate low.

Case 4 :- Train Network from scratch.

On Basis of Size of Data.

$D = \{x_i, y_i\}$
$|D|$ = size of train data

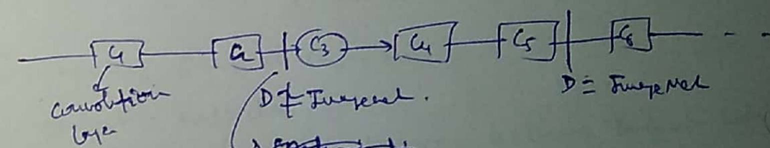
Case 1 :- $D \simeq$ ImageNet → data is similar to ImageNet and $|D|$ = small

use pretrained CNN Codes for feature engineering and then apply any ML algorithm.

Case 2 :- $|D|$ = large
$D \simeq$ ImageNet
finetune the complete Network (small learning rate)

Case 3 :- $|D|$ = medium sized, $D \simeq$ ImageNet } + finetune the last few layer.

Case 4 :- $|D|$ = small
$D \neq$ ImageNet



get + Fe + flattened → CNN code

Case 5 :- $|D|$ = large
$D \neq$ ImageNet
→ Initialize the model VGG16 + ImageNet
→ tune the complete model.

Reference: github: transfer learning CS231.

Unsupervised Learning

Clustering

In earlier lectures we understand supervised learning where we had features and levels.

$D = \{x_i, y_i\}$

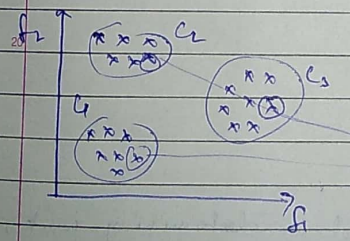
↳ $y_i \in \{0, 1\}$ → 2 class classification

$y_i \in R$ → → regression

But in Clustering,

we have features but no labels.

So with that features the task is to group/cluster 'similar' data points.



like in this figure a 5 year old kid can easily say, there point belongs to which cluster.

So, aim ⇒ group similar elements

↳ points are close together.

(a) points in a cluster is close together.

(b) points in different clusters are far away.

Here we will see clustering :- K means, Hierarchical clustering, DBSCAN in details.

Unsupervised learning

Clustering is also known as unsupervised learning.

Unsupervised because:

↳ here we don't have (y_i) to supervise the output.

Supervised $\{x_i, y_i\}$

↳ here we have y_i to supervise the output.

Semi-Supervised:

Here in small data set we have $\{x_i, y_i\}$ and for large data set we have only $\{x_i\}$

↳ this method is used when cost of obtaining y_i is large.

Application of Clustering

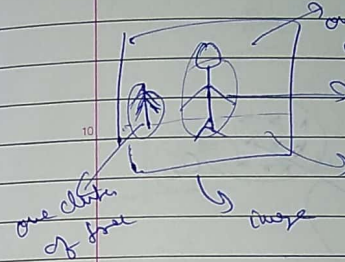
Earlier clustering was mainly used in data mining work, but in now a days a wide range of application of clustering is used in ML these days.

We have ecommerce company like, Amazon, Myntra, Flipkart, Ebay etc. who group similar customer based on their purchase behaviour.

Doing this it becomes easier for them to target a limited no. of people with offers that they might like.

It is also used in "image segmentation" (in computer vision)

↳ grouping/clustering similar pixels.



In amazon this is widely used because suppose if they wants to know the polarity of their products.

So the comments there can by millions, this could require a lot of man power to manually read the comments and assign a polarity.

So, one method is using unsupervised learning, we make clusters of polarity.

Now, to check the polarity, we take (10k) dataset of same polarity and will check in which cluster will it fit:

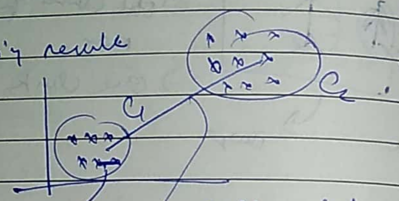
↳ those cluster it will fit we give same polarity to that whose cluster.

Here, analysing (10k) dataset become easier than (1M)

Metrics for Clustering

As we have seen in classification and regression we have some matrix to measure like, auc-score, log-loss etc. but these matrix require (y_i) or label, so can't be used in Clustering.

Geometry of a clustering result



Given: $D = \{x_i\}$

intra-cluster (→ small) :- with in a cluster

inter-cluster (large) :- across (or) b/w cluster

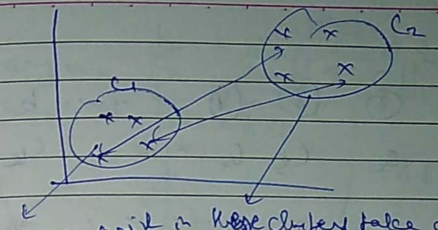
Here, as a matrix we will use Dunn-index.

for an ideal matrix :- inter-cluster dist → very high
intra-cluster dist → very low.

Dunn-matrix :- $D = \max_{i,j} \dfrac{d(i,j) \uparrow\uparrow}{\max_k d'(k) \downarrow\downarrow}$

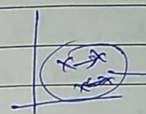
$d(i,j)$ → minimal intercluster distance
$d'(k)$ → intracluster distance

if D ↑ is high → good clustering



for every point in these clusters take distance

So, $d_1\ d_2\ d_3 \cdots$ ⇒ this should be high.



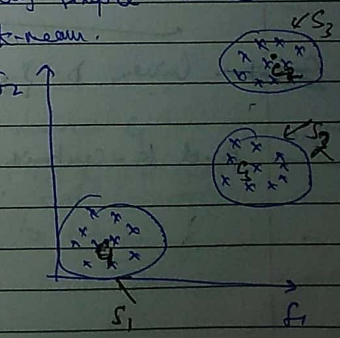
→ but dist b/w this point should be low.

⇒ for clustering we have many matrix to measure. but core concept of all is

→ low intra-cluster dist
→ high inter-cluster dist.

K-mean Geometric intuition.

Here, we are going to see very simple k-mean. although there is variants of k-mean.



Here C_1 C_2 C_3 are centroids. every point is closest to its centroids.

S_1 S_2 S_3 are sets of points

where, $S_1 \cup S_2 \cup S_3 = 0$, $S_1 \cap S_3 = \phi$
$S_1 \cap S_2 = \phi$ $S_3 \cap S_2 = \phi$

if we have

k clusters : (1) (2) (3) (4) ... k

then k-centroid : c_1 c_2 c_3 c_4 ... c_k

k-sets : S_1 S_2 S_3 S_4 ... S_k

$$c_i = \frac{1}{n_i} \sum_{x_j \in S_i} x_j$$ ← mean point to S_i

k-mean = centroid based clustering scheme.

Key, Big challenge is how to find best k.

it can be done by lloyd method.

Now, each point in sets is closest to its centroid than other centroid of other set.

K-Mean : Mathematical Formulation : objective Function

Given, $D = \{x_1, x_2 \dots x_n\}$

Task : find k-centroid : c_1 c_2 c_3 ... c_k

sets : S_1 S_2 S_3 ... S_k

Constraints
- $\forall i \; x_i \in S_j$
- $\forall i, j \; S_i \cap S_j = \phi$

$$\underset{c_1 c_2 c_3 \dots c_k}{\arg\min} \sum_{i=1}^{k} \sum_{x \in S_i} \|x - c_i\|^2$$

$\|x - c_i\|^2$ → square of dist of point to centroid should be low.

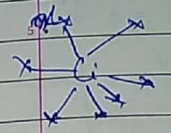
→ intra-cluster distance is minimized.

s.t $x \in S_i$

$S_i \cap S_j \neq \phi$ → Constraint.

$\sum_{x \in S_i} \|x - c_i\|^2$ → sum of sq. dist from centroid in cluster i

$\sum_{i=1}^{k}$ → all clusters.



This problem is v.v. hard to solve. because it takes time Complexity of (2^n) → so an approximation is made to solve this problem known as lloyd's algorithm.

K-Mean Algorithm. (Lloyd's Algorithm)

(1) First step is initialization. of k pts from D and call them centroids c_1 c_2 ... c_k

$D = \{x_1, x_2 \dots x_n\}$

(2) Second step is Assignment

→ for each point x_i in D

→ select the nearest c_j

dist(x_i, c_j) ⇒ small.

→ add x_i to set S_j

will continue in loop for each x_i

(3) Recompute centroids / update.

⇒ recalculate / update C_i's as follows.

$$C_i = \frac{1}{|S_i|} \sum_{x_i \in S_i} x_i$$

S_i → depth.

mean point

(4) Repeat step 2 & 3 until convergence.

Convergence will take place, when value of old centroids and new centroids are quite similar.

↳ $(C_k - C'_k) \ldots\ldots (C - C'_k)$

↳ approx same.

↳ for convergence.

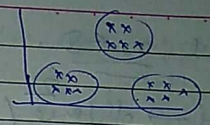
How to initialize K-means ++

Earlier we saw Lloyd's algorithm where we had to choose k-centroids randomly

But with random initialization there were sensitivity problem.

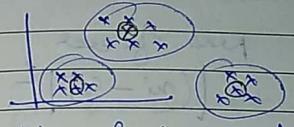
because each time we chose different sets of centroids, the cluster we get is different.

like here.



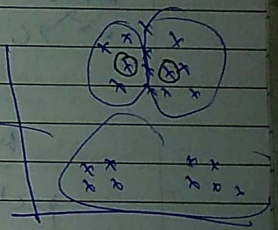
→ this is the cluster we want

if we initialize cluster like this



then cluster we get is same

if we initialize cluster like this



So the cluster we get is purely different.

So, one way to come up with this problem is

(1) repeat k-means multiple times with different initializations and pick the best clustering based on

↳ smaller intra dist and

↳ larger inter distance.

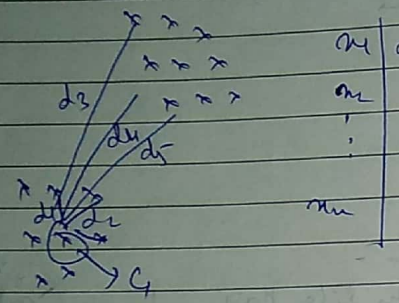
(2) ~~K-means++~~ K-means ++

So, here we initialize randomly but smartly.

(i) Pick the first centroid randomly → C_1 for $\forall x_i \in S$.

(2) Now for all $x_i \in D$.

Calculate $x_i \rightarrow$ distance^2 (x_i, nearest centroid)



x_1	d_1
x_2	d_2
:	
x_n	d_n

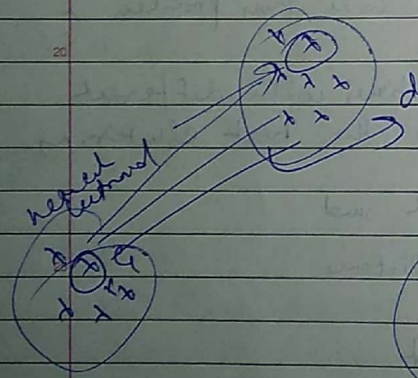
Now of each $|(x_i - c_1)|^2$

pick a pt. from D-{c_1} with a high probability proportional to d_i

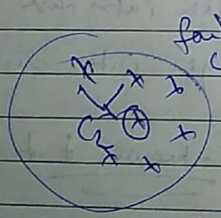
Repeat

pick those point whose distance from all centroid is reasonable

Done



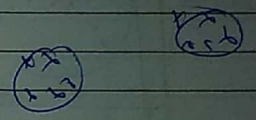
distance is reasonably close to so makes it as a centroid.



trying to pick pts → centroids that are as far as possible from other centroids that are already picked up.

this is a probabilistic approach.

By doing this we approach, we can't choose the farthest point as centroid, because this could be an outlier.



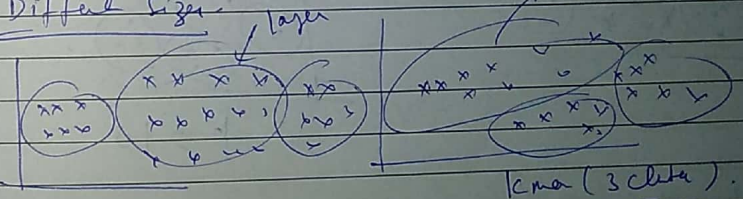
So k-mean is not affected by outlier.

Limitations of K-mean

⇒ K-mean has problem when clusters are of different
(1) Sizes
(2) Densities
(3) Non-globular shapes ⇒ non convex shapes.

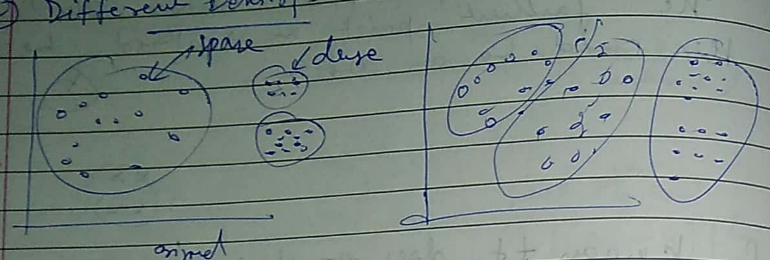
⇒ K-mean has problem when data contains outlier.

(1) Diffent Sizes.

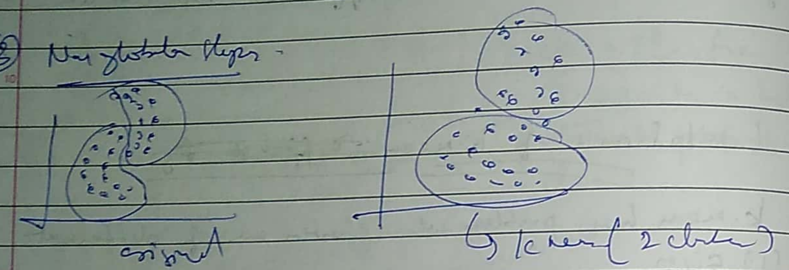


Here, when size changes. cluster set also changed, this should not happen.

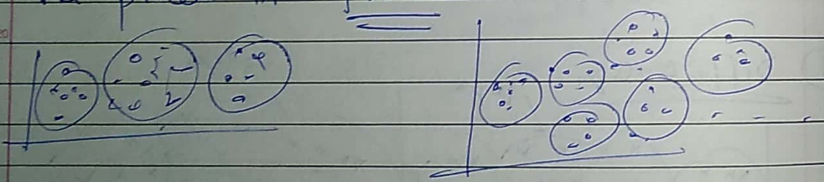
(2) Different Density -



(3) Non-globular shapes -



Here all can be reduced by dividing it into many cluster, and put it together, but this process is very hard.



K-Medoids

Earlier in k-mean, for taking centroids, we were taking mean of data points in each set.

But here the core idea is to take any point as centroid from particular set.

Like if we are vectorizing text data, | w1 | w2 | w3 | w4 | ...

But lets say, if we take w4 as centroid then this would be much useful.

So, if we take avg. value, then this might not fall in dataset.

because we can read the review from (w4) interpretability becomes easier.

Big-idea :- What if each centroid is a datapoint in D
$c_i = x_i \in D$
⇓
k-medoids

Partioning around Mediods (PAM) : k medoids.

(1) Initialization :- Similar as k-mean++ → probabilistic method pick k pts from D.

(2) Assignment :- closest medoid
{ $x_i \in S_j$ if medoid is the closest medoid to x_i }

(3) Update / recompute

in k-mean centroid update function was.

↳ $c_j = \frac{1}{|S_j|} \sum_{x_i \in C_j} x_i$

But in

k-medoids.
(PAM).

(a) we swap each medoid with a non-medoid points!

(b) if loss decreases keep the swap else reassign the previous mediod.

eg: (x_1) x_2 x_3 x_4 x_5 (x_6) x_7 x_8 x_9 x_{10}

then we have two medoids x_1 x_6
rest are non medoids.

loss in k-mean.
↓ min^m

↳ $\sum_{i=1}^{k} \sum_{x \in S_i} ||x - m_i||^2$
↳ medoid

if we take x_2 as medoid
and x_1 as non medoid

(a) loss - value $x_1 = M_1$; $x_6 = M_2 \rightarrow l_1$

(b) Swap $M_1 = x_2$
$M_2 = 6$
↳ loss - value = l_2

if $(l_2 < l_1)$ then $M_1 = x_2$ and $M_2 = x_6$

else $M_1 = x_1$ and $M_2 = x_6$
↳ reassign previous value and further swap with next data point.

Here we are using dist (x_i, M_j) → sim (x_i, x_j)
medoid data pair
or dist $(x_i\, x_j)$ → kernel matrix.

true in medoids since it uses distance approach
so, we can even solve problem from similarity (x_i, x_j)

So, with k-medoids
↳ interpretability ↑
↳ kernelization, similarity, dist.

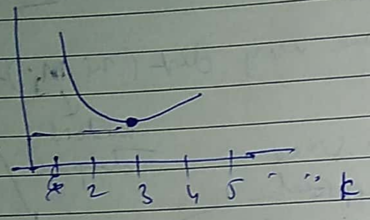
Determining the right k.

(1) Domain knowledge
↳ like in amazon food reviews we already know that review is +ve or -ve.

② "Elbow Method" or Knee-method

$loss = \sum_{i=1}^{k} \sum_{x_i \in S_i} ||x - c_i||^2 \rightarrow$ minimize

Here we plot b/w loss and various k, and we choose k with min^m loss.



k-means Sklearn Documentation.

[For k-medoids use different package code kmedoids.py.

Time and Space Complexity.

Time Complexity $O(nkdi)$
(#pts, #cluster, #iteration)

Usually $k < 10$, $i \leq 300$

$\approx O(nd)$ → linear time.

Space $O(nd + kd)$ (store i/p pts, store k pts)
$\approx O(nd)$ (linear)

Hierarchical Clustering Technique.

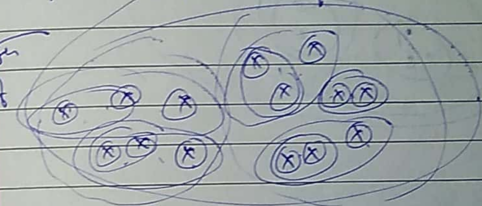
Agglomerative & Divisive, Dendogram.

Hierarchical clustering is divided into two part
→ Agglomerative &
→ Divisive.

Agglomerative → popular.

initially it consider all the points as cluster. Then, combine the two closest points or cluster, then further repeat the process.

They cluster on basis of sim/dist

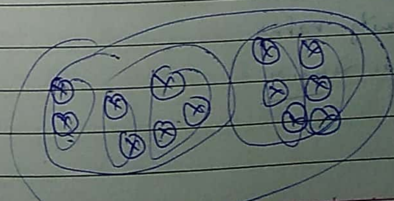


Step 1 → 14 cluster.
Step 2 → 9 cluster.
Step 3 → 5 cluster.
Step 4 → 2 cluster.
Step 5 → 1 cluster.

initially each point is cluster

So, each time no. of cluster is decreasing.

Devisive ⇒ it is pretty hard to implement & so less used. it consider all points in 1 cluster, then it further seperates into different ~~different~~ cluster.



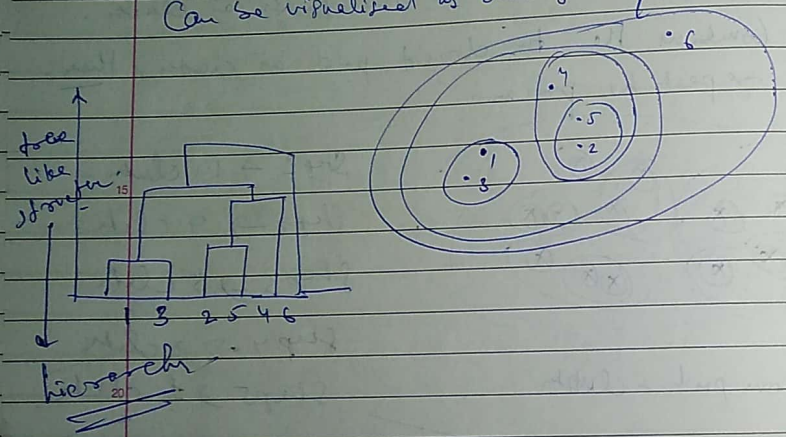
① 1-cluster.
② 2-cluster.
③ 5-cluster.
④ 13-cluster.

In kmeans we had to rerun the process to get diffent no. of clusters.

But here in these two algo. we don't need to rerun the whole algo. We can stop at any cluster we want.

Algorithm

Can be visualised as dendrogram (like trees)

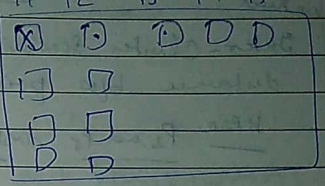


Agglomerative Clustering

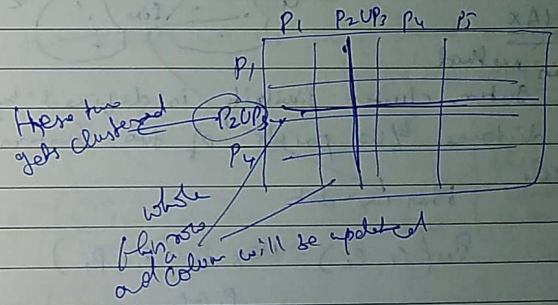
1. Compute the proximity matrix
2. Let each data point be a cluster
3. Repeat
4. Merge the two closest cluster.
5. Update the proximity matrix
6. Until only a single cluster remains.

- Key operation is the computation of the proximity of two clusters.

initially,
for each pair of cluster we calculate proximity of two clust.



after that if two clust combined then we calculate proximity of that new cluster to other clust.



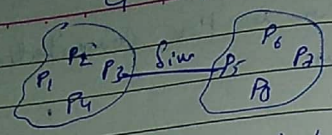
proximity is calculated by sim/dist b/w 2 clusters

Proximity Methods: Advantages and limitation.

How to define inter cluster similarity?

⇒ By ⇒ ~~MIN dist~~
⇒ MAX dist
⇒ Group avg
⇒ Distance b/w Centroids
⇒ Other methods driven by our objective.

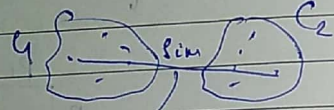
MIN



In this method
Inter-cluster similarity is defined by the min^m distance b/w points of two clusters.

Here p_3 and p_5 distance.

$$Sim(C_1, C_2) = \min_{p_i \in C_1,\ p_j \in C_2} Sim(p_i, p_j)$$



MAX

In this method
Inter-cluster similarity is defined by the max^m distance b/w points of two clusters

Here, this max^m dist

$$Sim(C_1, C_2) = \max_{p_i \in C_1,\ p_j \in C_2} Sim(p_i, p_j)$$

Group Avg

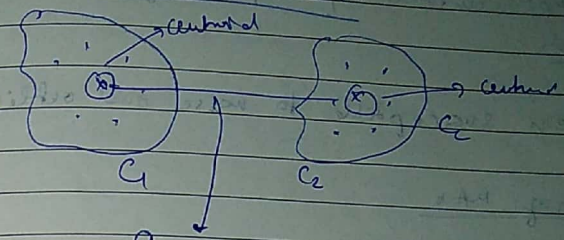
Here, Similarity b/w two group is clarified by avg. dist b/w points of two cluster.

$$Sim(C_1, C_2) = \frac{\sum_{p_i \in C_1,\ p_j \in C_2} Sim(p_i, p_j)}{|C_1| * |C_2|}$$

Size of two cluster.

Blog :- cs.wmich.edu / clustering analysis

Distance b/w Centroids

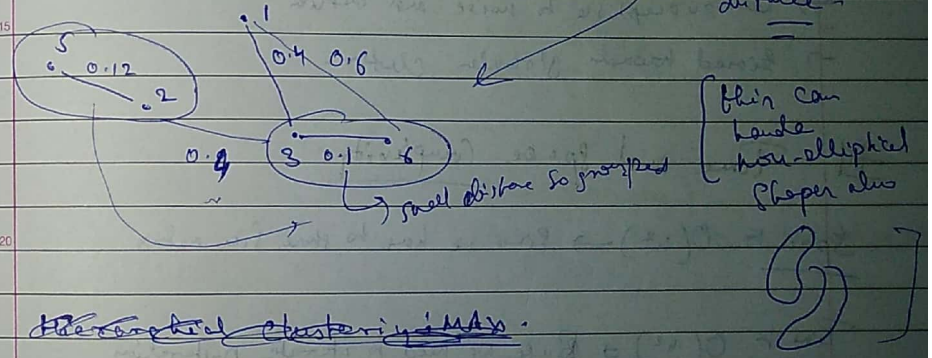


Sim depend on dist b/w centroids.

This method is not extensively used.

Hierarchical Clustering : MIN

Here two points will be group based on min^m distance.



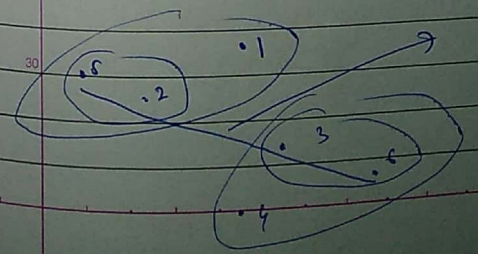
This can handle non-elliptical shaped also

~~Hierarchical clustering : MAX~~

Limitation of MIN

→ Sensitivity to noise and outliers.

Hierarchical Clustering : MAX



Some dist b/w pt. 5 and 6 is lager than dist b/w pt. 5 and 1 and 6 and 4.

∴ (5, 2, 3 & 6) can't be in same cluster.

Here, Max distance b/w points of cluster is considered.

⇒ It is less susceptible to noise and outlier.

Limitations of MAX

⇒ tends to break large cluster.

⇒ Biased towards globular cluster.

Group Avg

↳ It is a compromise b/w (Single → MIN dist) and (Complete → MAX dist) like.

⇒ less susceptible to noise and outlier

⇒ Biased towards globular cluster.

Time and Space Complexity.

Space :- $O(n^2)$ → Since we have to store sim. matrix

Time :- $O(n^2)$ ⇒ Since we need to iterate n-iterations to group 2 clusters.

and update sim-matrix takes ~ $O(n^2)$

$\therefore O(n \times n^2) = O(n^3)$

Limitations of Hierarchical clustering.

(1) No main objective function that we are directly solving :-

(2) for MIN → outlier problem

MAX → breaks large cluster

↳ can't accomodate different sized cluster.

(3) Most imp problem

↳ Space and time Complexity very large

↓ ↓

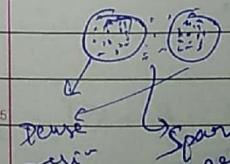
$O(n^2)$ $O(n^3)$

sklearn.cluster.AgglomerativeClustering( ) .

(3) DBSCAN (Density Based Clustering)

↳ Density Based Spatial Clustering of applications with noise.

Density Based Clustering.



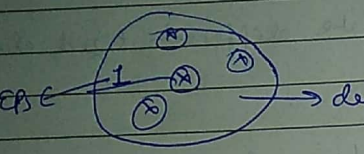
In density based clustering, it seperates two dense region if there is a sparse region in it.

MinPts and Eps: Density.

↳ These are the hyperparameters of "DBSCAN".

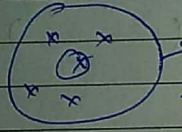
(1) density at a P: no. of point within a hypersphere of radius (Eps) around p

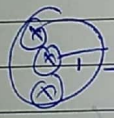
↳ Circle/Sphere



(2) 'dense' region: a hypersphere/circle of radius (Eps) that contains atleast MinPts points.

↳ say 5 ↳ say 1

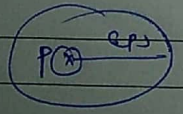




Core point, Border point & noise point

Given, D = {xi} and Given, MinPts, Eps.

(1) Core point: if p has ≥ MinPts points, in an Eps radius around it (P)



Core point always belong to dense region.

(2) Border pt (p)

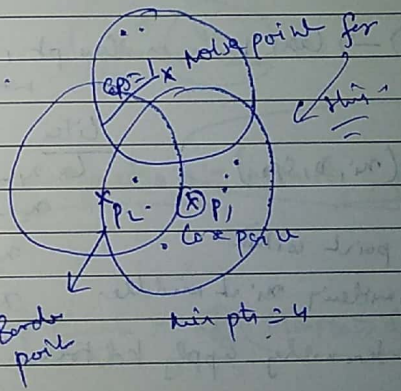
(1) P is not a core point ⇒ P has < minpt point in Eps radius and

(2) P ∈ Neighbourhood (Q) ↳ core point

dist(p,q) ≤ Eps.

(3) Noise point:

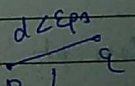
↳ neither Core pt nor a border pt.



Density Edge and Density Connected points:

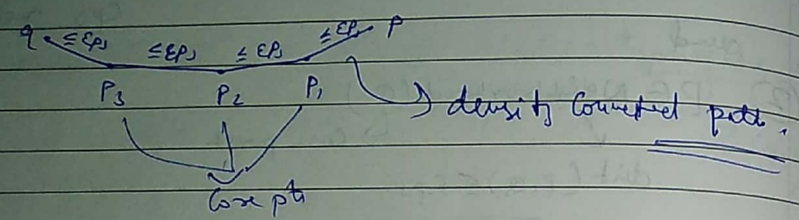
(1) density edge: → connection

↳ if p, q → core pts and dist(p,q) ≤ Eps



(2) Density Connected pts :-

if p, q are → core pts.



DBSCAN Algorithm.

Step 1 for all ($x_i \in D$) → label them as core pt, border pt, noise pt.

S_i = rangeQuery(x_i, D, eps)

↓

this will give every point within a circular radius eps, considering x_i at middle.

↳ this internally apply kd tree

like
↳ x_1 → core
x_2 → border
x_3 → noise
x_4 → core
⋮

Step 2

remove all noise pts from your data

↳ sparse region ∃ doesn't belong to any clustering.

Step 3

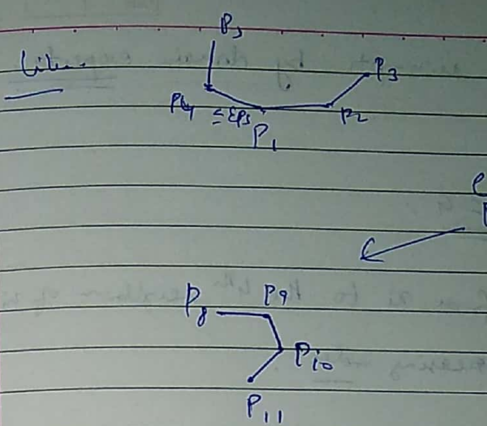
For each core pt 'p' not assigned to a cluster

(a) create a new cluster with p

(b) add all pts that are density connected to p into this new cluster

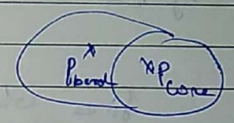
this can be achieved by range query (...)

like



if point P8 is not in this density range then create new cluster with point P8.

(4) each border pt → assign it to the nearest core pt's cluster.



Hyperparameter :- Minpts, Eps.

(1) Minpts :-

(a) rule of thumb :- (Minpts ≥ d+1) ← dimensionality

$x_i \in R^d$ → no. of dimension

Typically; Minpts ≈ 2*d

(b) if larger value of Minpts → dataset is more (noisy) → remove noise pts.

(p Eps) ≤ Min pts.

if within Eps radius from p if no. of pts < min pts then, p is consider to be noise point.

(c) DBR can choose min-pts by domain experts.

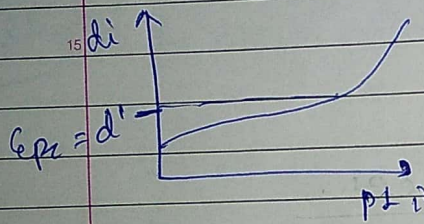
(2) Eps let min-pts = 4.

(a) $x_i \rightarrow d_i$: dist from x_i to the 4th neighbour of x_i

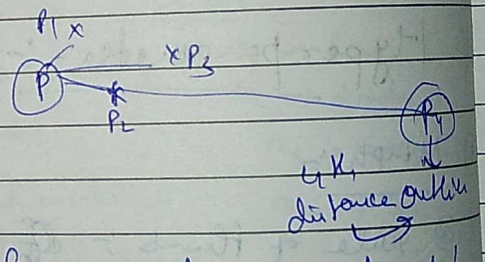
(b) sort d_i's in increasing order.

x_4 d_4 1
x_2 d_2 1
!

Now, Select Eps by Elbow method.



Since when d_i = high



So P is also considered to be an outlier (noise)

Advantage and Limitation of DBSCAN.