24/5/18

Machine Learning | Coursera | Stanford

* WEEK 1 :

Click stream data : web click data being collected for ML purpose.

Note SVM allows the computer to deal with infinite number of features.

e.g. of unsupervised learning - (Cocktail party problem algorithm)

Microphone recordings with two people speaking simultaneous. The unsupervised algo will distinguish the two voices as the diff microphone recordings have diff. amplitudes of the two speaking people thus giving two separate outputs with individual 1 speaking in recording 1 as output 1 and output 2 with individual 2's voice separated out.

Training Set
↓
Learning Algo
↓
I/P → [h] → O/P
(hypothesis)

* f^N to map i/p to corresponding o/p

Hypothesis is the standard terminology used by people in ML

e.g. $h_\theta(x) = \theta_0 + \theta_1 x$ [Linear Regression]
(parameters of the model)

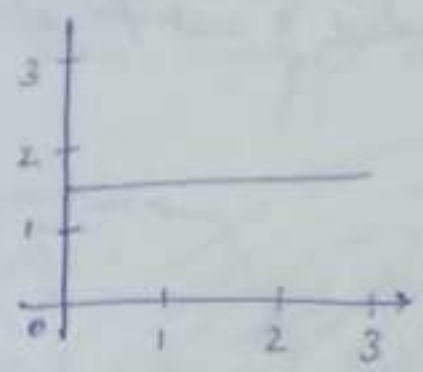
MODEL

(fitting linear f^N on the training dataset

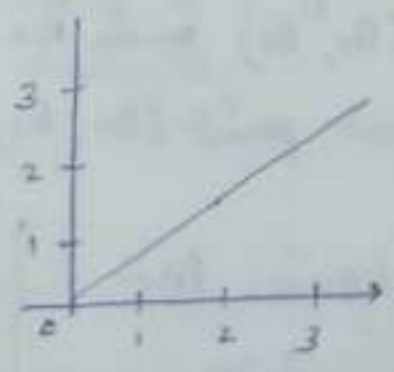
Linear regression ≡ one variable

(1) = Univariate Linear Regression

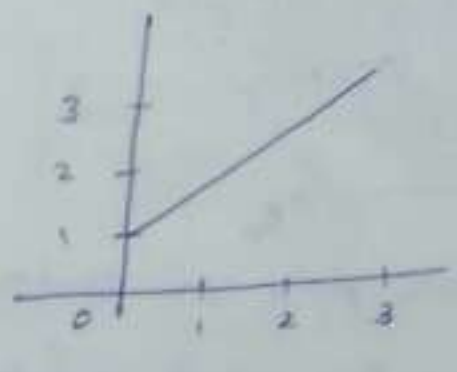
Diff θ_0 and θ_1 lead to different hypothesis f^N, $h_\theta(x)$.



$\theta_0 = 1.5$
$\theta_1 = 0$



$\theta_0 = 0$
$\theta_1 = 0.5$



$\theta_0 = 1$
$\theta_1 = 0.5$

for $h_\theta(x)$
$= \theta_0 + \theta_1 x$.

We choose θ_0, θ_1 so that $h_\theta(x)$ is close to y for our training eg. (x, y)

minimize θ_0, θ_1 $\frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$ i.e finding θ_0 and θ_1 to minimize this expression.

where m = no. of training examples.

overall objective fn for linear regression.

i.e minimize $J(\theta_0, \theta_1)$
θ_0, θ_1

where $J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$ is our goal to minimize.

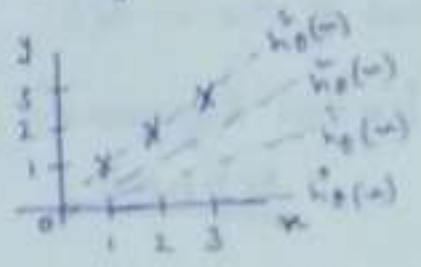
↓ cost fn for linear regression

(Squared error fn)

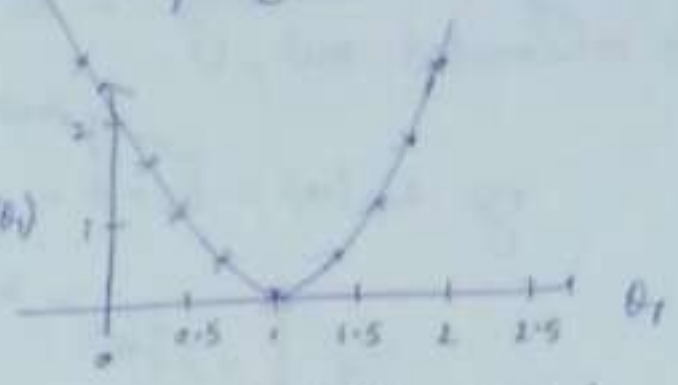
$h_\theta(x)$: for fixed θ_0 and θ_1, this is a fn of x.

$J(\theta)$ :: fn of θ_0 and θ_1.

e.g.



(training example)



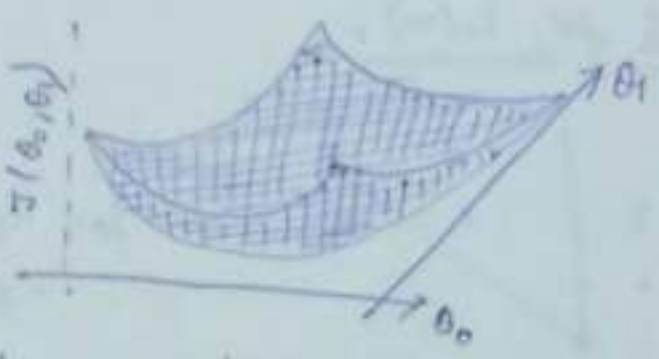
(cost fn values)

(assuming $\theta_0 = 0$)

Each value of θ_1 corresponds to a different hypothesis $h(x)$ or to a different straight line fit on the left and for each value of θ_1, different value of $J(\theta_1)$ can be derived.

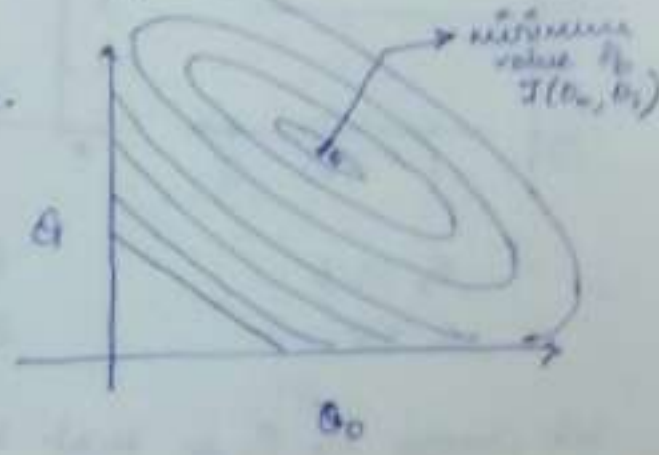
Above $J(\theta)$ graph $\Rightarrow \theta_1 = 1$ is the best parameter value giving min $J(\theta_1)$

when both θ_0 and θ_1 are considered, $J(\theta_0, \theta_1)$ is a 3D plot like:



The height of the graph at certain (θ_0, θ_1) gives the value of cost fn at that point (θ_0, θ_1)

We will use contour plots or contour figures like.



The ovals show the set of points that takes on the same value for $J(\theta_0, \theta_1)$. Points lying on the same oval have the same value.

O gives (θ_0, θ_1) for global minimum.

So, we need an algo to automatically find value of θ_0 and θ_1 that minimizes cost fn J. It is called gradient descent. It is used to minimize other fn as well not just cost fn J for linear regression. Also called batch gradient decent.

25/5/18

In this algo, we start ~ some θ_0, θ_1 (say 0) and keep changing θ_0, θ_1 to reduce $J(\theta_0, \theta_1)$ until we get minimum or local minimum. Gradient decent produces different minimum based on different starting points.

repeat until convergence: {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad (\text{for } j=0 \text{ and } j=1)$$

}

Step1: temp 0 := $\theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$ → assignment operator (assigns RHS to LHS)

Step2: temp 1 := $\theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$ → derivative term giving the slope.

Step3: θ_0 := temp0

Step4: θ_1 := temp 1

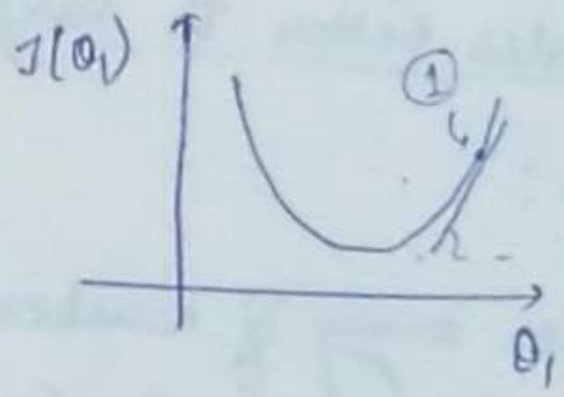
→ learning rate (decides how big step to be taken while descending).

Can't do step3 before step 2 bcoz otherwise in $\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$ updated θ_0 would be used. Hence, we need simultaneous update of θ_0 and θ_1.

while a=b is truth assertion (simply asserts that a=b)

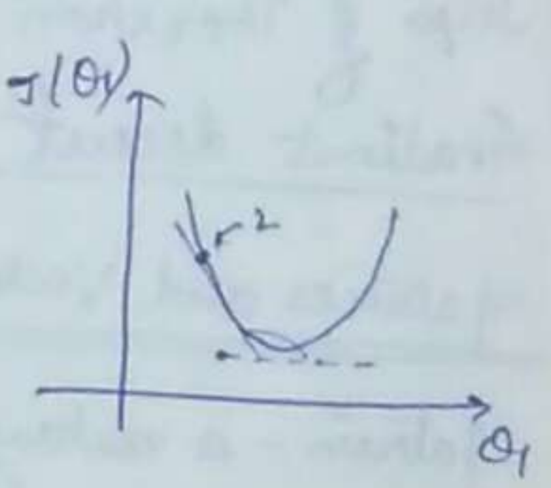
Hence, we can write a := a+1 but not a = a+1.

Note If α is too small, gradient decent can be slow, if its too large it can overshoot the minimum. Hence failing to even converge or diverge.



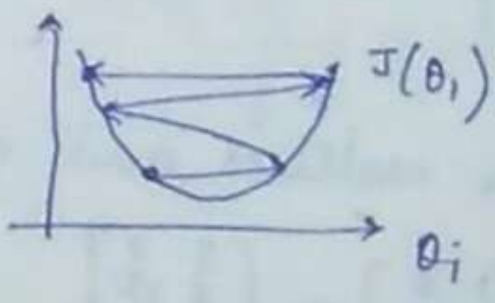
e.g. 1

In this case θ_1 will ↓ bcoz of 1 being the starting point.



e.g. 2

In this case θ_1 will ↑ bcoz of 2 being the starting point.



(in case of high α)

If θ_1 is initially at local minimum, gradient decent will leave θ_1 changed to nothing different but original θ_1, only as $\frac{d}{d\theta_1} J(\theta_1) = 0$.

(imp)

Gradient decent can converge to a local minima, even ~ α fixed bcoz gradient " will automatically take smaller steps and so no need to ↓ α over time.

Applying gradient descent to minimize squared error cost fn.

$$\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})$$

$$\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x^{(i)} \quad \text{as } h_\theta(x^{(i)}) = \theta_0 + \theta_1 x^{(i)} \rightarrow \text{hypothesis fn}$$

and we know: $\theta_j = \theta_j - \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad \{j=0, 1\}$.

Convex fn: a bow shaped fn and so has just one global optimum instead of having any local optima.

The above algorithm is also called 'Batch Gradient Descent'. It refers to the fact that we've looking at all of the training examples. So, in gradient descent when computing the derivatives we are computing the sums over our 'm' training examples and so term refers to the fact that we are looking at the entire batch of training examples.

There are methods that solves for $\min J(\theta_0, \theta_1)$ without needing multiple steps of iteration like in gradient descent.

Gradient descent scales better to larger data sets than normal eqⁿ method.

Matrices and Vectors:

Matrix - a rectangular array of numbers. e.g. $\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$ is $\mathbb{R}^{4\times 2}$

Dimension of matrix - (no. of rows) × (no. of columns)

Vector - An $(n\times 1)$ matrix i.e matrix with just 1 column. e.g. $\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$ is a 4-D vector $= \mathbb{R}^4$

1-indexed vector; 0-indexed vector (starting indexing from zero)

A, B, C, X, etc used for matrices and a, b, c, x for vectors representation commonly.

Scalar multiplication: $3 \times \begin{bmatrix} 1 & 0 \\ 2 & 5 \\ 3 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 0 \\ 6 & 15 \\ 9 & 3 \end{bmatrix}$ (it is commutative)

e.g. $h_\theta(x) = -40 + 0.25x$ & $x = \begin{bmatrix} 21 \\ 14 \\ 15 \\ 85 \end{bmatrix}$

Hence $h_\theta(x)$ can be calculated as: $\begin{bmatrix} 1 & 21 \\ 1 & 14 \\ 1 & 15 \\ 1 & 85 \end{bmatrix} \times \begin{bmatrix} -40 \\ 0.25 \end{bmatrix}$ (in one go)

∴ **Prediction = Data Matrix * Parameters** (more computationally efficient.)

Matrix matrix multiplication is a way to the gradient descent iterative approach for calculating θ_0 and θ_1.

Note $A \times B \neq B \times A$ (Matrix multiplication is not commutative)

$(A \times B) \times C = A \times (B \times C)$ (. . . associative)

$I_{n \times n}$: Identity matrix of dimension $n \times n$. (has 1s along the diagonals) $\begin{bmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{bmatrix}$

$A \times B = B \times A$ when B is Identity matrix (but of diff. dimension)

Matrix inverse : If A is an $m \times m$ matrix, and if it has an inverse,

$A(A^{-1}) = A^{-1}A = I$ (If A is a square matrix and only square matrices have inverses)

Just like 0, matrix A if a zero matrix has no inverse. In ML, matrices not having inverses can be considered close-to zero.

Singular / Degenerate Matrix : Matrices that don't have an inverse.

If $B = A^T$ then $B_{ij} = A_{ji}$

29/5/18

Multivariate Linear Regression:

$x^{(i)}$ = input (features) of i^{th} training example.

$x_j^{(i)}$ = value of feature j in i^{th} " " . e.g. $x^{(2)} = \begin{bmatrix} 1416 \\ 3 \\ 2 \\ 40 \end{bmatrix} \in \mathbb{R}^4$ if $n=4$

n = no. of features.

Now, $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$ $\{x_0^{(i)} = 1\}$ → = adding an additional 0 feature

NOW, $X = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n+1}$ $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} \in \mathbb{R}^{n+1}$

$h_\theta(x) = \theta^T X$ $\{[(n+1)\times 1] * [1 \times (n+1)]\}$

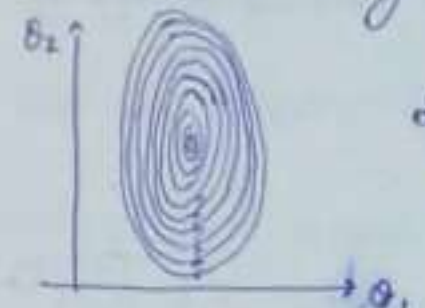
$J(\theta_0, \theta_1, \ldots \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 = J(\theta)$

Gradient descent

Repeat $\{ \theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \}$

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$ { simultaneously update θ_j for $j = 0, \ldots, n$)

Note Gradient decent converges more quickly when different features take on similar ranges of values.



In this case when θ_1 and θ_2 are in different ranges, gradient decent has a hard time reaching the global minima.

If x_1 and x_2 i.e features are in same range:

It is easier for gradient decent to reach global minimum.

for this mean normalization can be done i.e replacing x_i with $\frac{x_i - \mu_i}{s_i}$ to make features have approx 0 mean. (s_i → (range of values) i.e max-min) } feature scaling

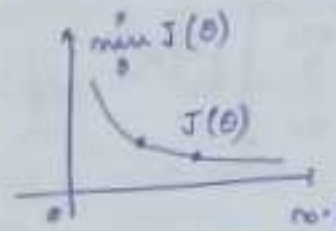
eg. no. of bedrooms

1	$\frac{1-2}{5} = -0.2$
2	$\frac{2-2}{5} = 0$
3	$\frac{3-2}{5} = 0.2$
4	$\frac{4-2}{5} = 0.4$
5	$\frac{5-2}{5} = 0.6$

} normalized values.

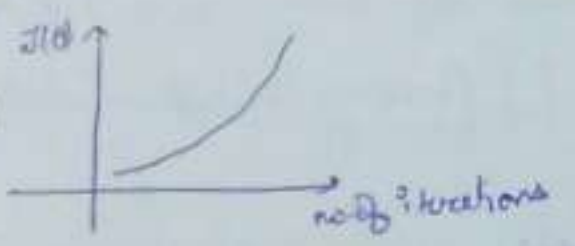
3/5/18

This is called feature scaling to converge gradient decent in much lesser iterations. Gradient decent basically finds θ for which $J(\theta)$ is minimum.



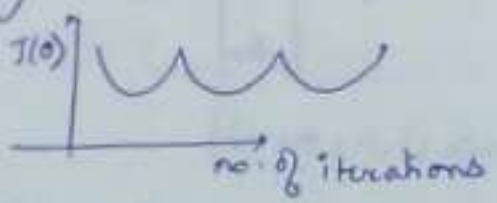
When gradient decent work properly, $J(\theta)$ dec with every iteration. The plot helps us judge if the gradient decent is converging. Automatic convergence test also helps us do so by checking if $J(\theta)$ dec by less than ϵ in 1 iteration. but determining this ϵ for different applica is difficult

(i)



⇒ gradient decent working improperly.
⇒ need to ↓ α.

(ii)



⇒ need to use smaller α.

Note: α should not be too small, as then we would need many itera to get minimum θ.

Note If α is too large, slow convergence is possible but $J(\theta)$ may not ↓ on every iteration.

So, best way is to plot $J(\theta)$ for different values of α against the no. of iterations. α that leads to convergence of $J(\theta)$ should be selected.

→ Polynomial regression :

eg. $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$ (cubic fit to my data)

$x_1 \quad x_2 \quad x_3$

features.

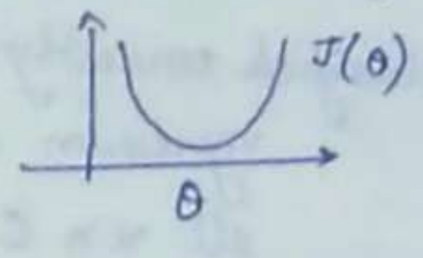
eg. $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 \sqrt{x}$

- Normal eqn ↓ – an alternate to gradient decent :

method to solve for θ analytically, so that θ can be obtained in 1 go instead of repetitive iterations as in gradient decent.

eg. if 1D ($\theta \in R$)

$J(\theta) = a\theta^2 + b\theta + c$



$\frac{d}{d\theta} J(\theta) = 0 \Rightarrow$ solve for θ.

in general : $\frac{\partial}{\partial \theta_j} J(\theta) = 0$ (for every j)

and solve for $\theta_0, \theta_1, \ldots \theta_n$.

$$\theta = (X^T X)^{-1} X^T y$$

↳ this gives θ to minimize the value of cost fn.

X : $m \times (n+1)$ dimensional vector

Y : m dimensional vector

$$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix} \in R^{n+1}, \quad \text{so, } X = \begin{bmatrix} —(x^{(1)})^T— \\ —(x^{(2)})^T— \\ \vdots \\ —(x^{(m)})^T— \end{bmatrix} = \text{design matrix.}$$

eg. if $x^{(i)} = \begin{bmatrix} 1 \\ x_1^{(i)} \end{bmatrix}$, $X = \begin{bmatrix} 1 & x_1^{(1)} \\ 1 & x_2^{(i)} \\ \vdots & \vdots \\ 1 & x_m^{(i)} \end{bmatrix}_{(m \times 2)}$ and $y = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix}$

Feature scaling isn't necessary when we use Normal eqn.

Gradient Decent	Normal eqn
i) Need to chose α	i) No need to chose α.
ii) Many iterations needed	ii) Don't need to iterate.
iii) works well even when 'n' (features) is large.	iii) is slow when n is large bcoz $(X^T X)^{-1}$ is $(n \times n)$ matrix.
iv) when $n = 10^4$, we switch to gradient decent.	iv) preferred upto atleast $n = 1000$

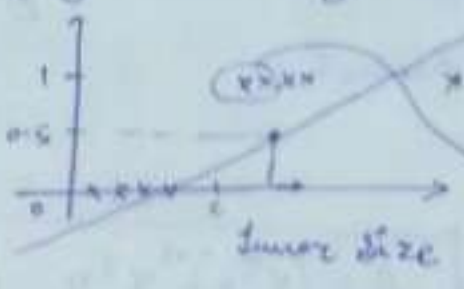
If X^TX is singular or non-invertible matrix, even then the θ value can be computed. This condition of non-invertibility generally arises when there are repeated features in x, or when $m \leq n$, i.e there are less training examples than there are no. of features. In this case either we delete some features or use regularization which will help fit parameters even when we have a small training set.

→ Logistic Regression: $\boxed{0 \leq h_\theta(x) \leq 1}$

for binary classification problems - $h_\theta(x) \geq 0.5$, predict $y = 1$
< 0.5, " $y = 0$.

applying linear regression to a classification problem isn't a good idea.

e.g.



→ linear regression line.

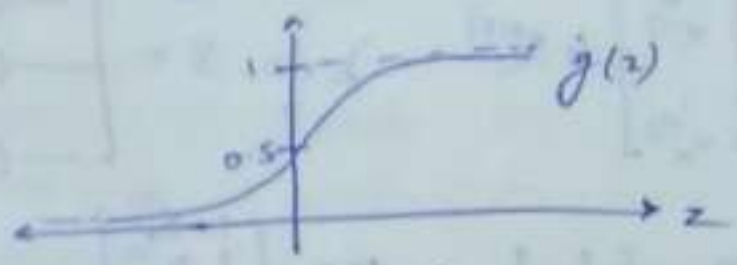
→ not classified correctly when linear regression used even though $y = 1$ for all $x > c$ and $y = 0$ for all $x < c$.

for logistic regression -

$h_\theta(x) = g(\theta^T x)$

where, $g(z) = \frac{1}{1+e^{-z}}$ (sigmoid fn/ logistic fn)

$\therefore \boxed{h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}}$



Also satisfying $0 < h_\theta(x) < 1$

estimated probability that $y = 1$ on input x.

e.g. if $x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1 \\ \text{tumor size} \end{bmatrix}$

$h_\theta(x) = 0.7$

⇒ there are 70% chance of tumor being malignant.

$\boxed{h_\theta(x) = P(y=1 \mid x; \theta)}$ - probability that $y = 1$, given x, parameterized by θ.

$y = 0$ or 1

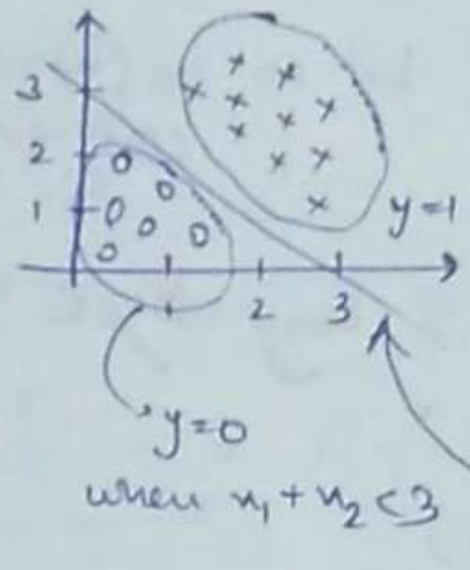
So, $P(y=0 \mid x, \theta) + P(y=1 \mid x; \theta) = 1$

when $z \geq 0$, $g(z) \geq 0.5$.

i.e whenever, $\boxed{\theta^T x \geq 0,\ h_\theta(x) = g(\theta^T x) \geq 0.5}$

So, $y=1$ when $\theta^T x \geq 0$ and $y=0$ when $\theta^T x < 0$.

eg.



$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$

Suppose, we chose $\theta = \begin{bmatrix} -3 \\ 1 \\ 1 \end{bmatrix}$, then

$y=1$ if $-3 + x_1 + x_2 \geq 0$

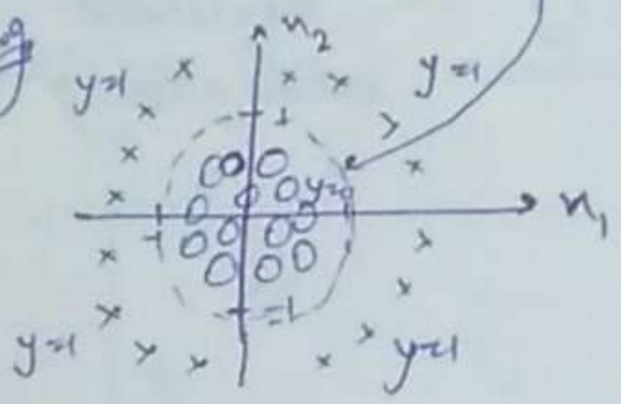
$x_1 + x_2 \geq 3$

$\boxed{x_1 + x_2 = 3}$ → 'DECISION BOUNDARY' that divides the region of $y=0$ & $y=1$.

↳ a property of the hypothesis and the parameter $\theta_0, \theta_1 \ldots \theta_n$ and not a property of the dataset.

Non-linear decision boundaries :

eg

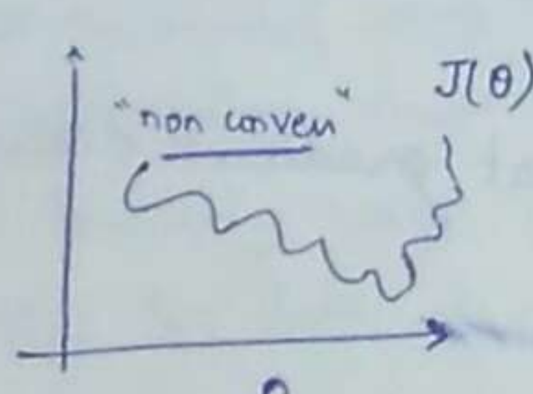


$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2)$ Let $\theta = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$

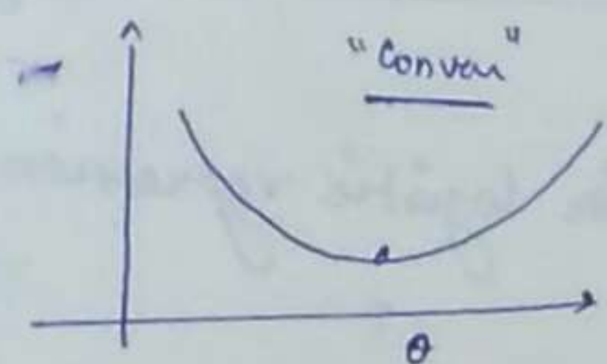
So, 'y=1' if $-1 + x_1^2 + x_2^2 \geq 0$

$\boxed{x_1^2 + x_2^2 \geq 1}$ → Decision boundary

Training set does not define the decision boundary but only used to fit the parameter θ, which in turn decides the decision boundary.



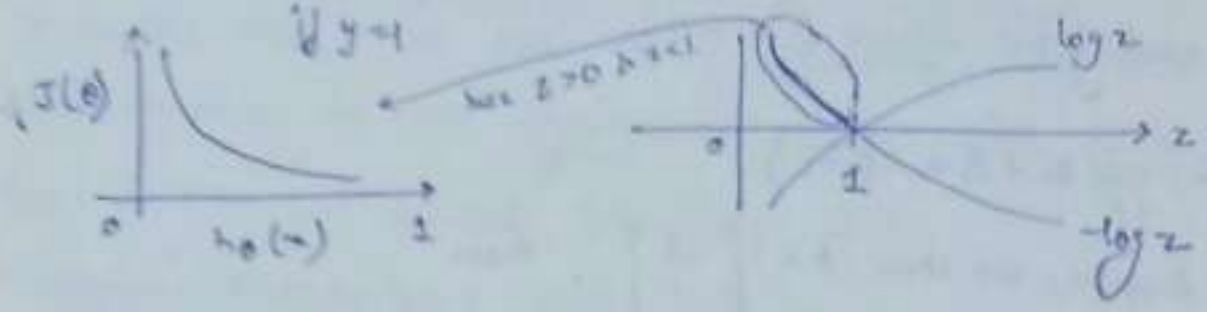
In this case, it is not sure that gradient descent converges to global minimum.



Here, gradient descent will always converge to global minimum.

$J(\theta) = \frac{1}{2}(h_\theta(x) - y)^2$ ends up to be non-convex for $h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$ and so, we define new $J(\theta)$ for logistic regression.

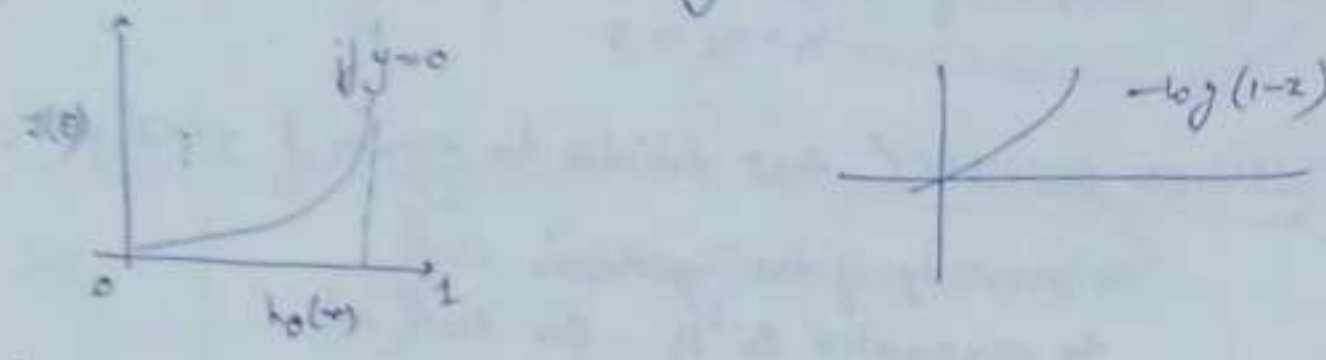
$J(\theta) = \begin{cases} -\log(h_\theta(x)) & \text{if } y=1 \\ -\log(1-h_\theta(x)) & \text{if } y=0 \end{cases}$ for single training example.



So, cost = 0 if $y=1$, $h_\theta(x)=1$

but as $h_\theta(x) \to 0$ cost $\to \infty$

If $h_\theta(x) \to 0$, but $y=1$, learning algo will be penalized by a very large cost.



$$\boxed{\text{cost}(h_\theta(x), y) = -y\log(h_\theta(x)) - (1-y)\log(1-h_\theta(x))} \checkmark$$

$$\therefore J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)}\log h_\theta(x^{(i)}) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right]$$ { maximum likelihood estimation }

(it is convex)

$\min_\theta J(\theta)$:

Repeat {

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \quad \text{——— (1)}$$

}

(Simultaneously update all θ_j)

(1) is same as we got for linear regression. So, are linear and logistic same or different algos. The difference is for linear regression $h_\theta(x) = \theta^T x$ but for logistic $h_\theta(x) = g(\theta^T x)$.

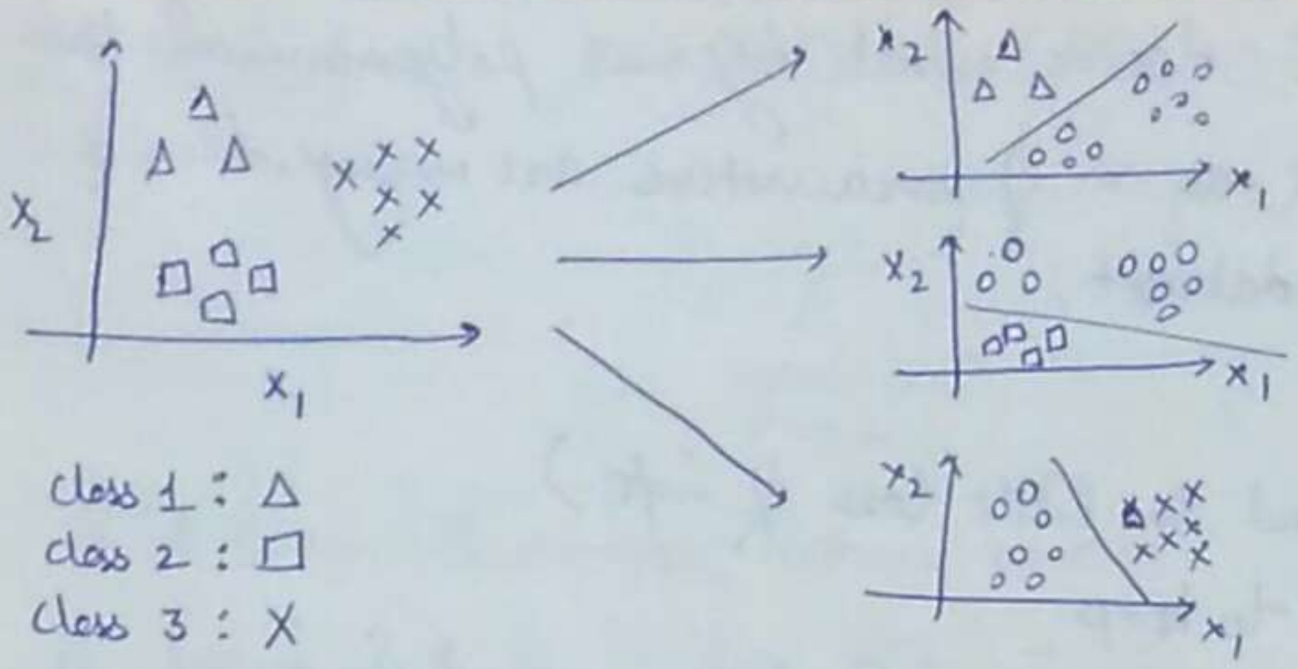
feature scaling can be done for logistic regression also so that gradient descent converges fast.

⇒ Optimization algorithm: complex but faster than gradient descent. α need not be chosen. e.g. conjugate gradient descent, BFGS, L-BFGS.

→ Multiclass classification: One v/s All. or On v/s Rest

for this we turn the problem into three separate 2 class classification problems

e.g.



$h_\theta^{(1)}(x)$ (△s as +ve class)

$h_\theta^{(2)}(x)$ (□s as +ve class)

$h_\theta^{(3)}(x)$ (X as +ve class)

→ denotes the class.

So, we have fit 3 classifiers and each one is trained to recognize 1 of 3 classes

$h_\theta^{(i)}(x) = P(y=i|x;\theta)$ $(i=1,2,3)$

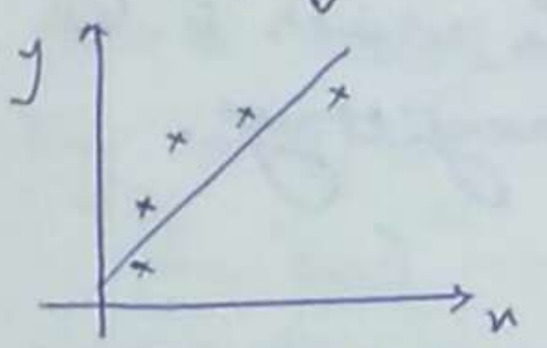
↓
logistic regression classifier

On a new input x, to make a predic^n, pick the class i that maximizes

$\max_i h_\theta^{(i)}(x)$: whichever value of i gives maximum value of $h_\theta(x)$ is predicted to be the output with max confidence.

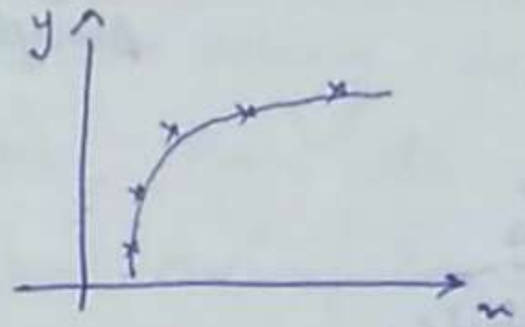
31/5/18

→ Overfitting: causes poor performance.

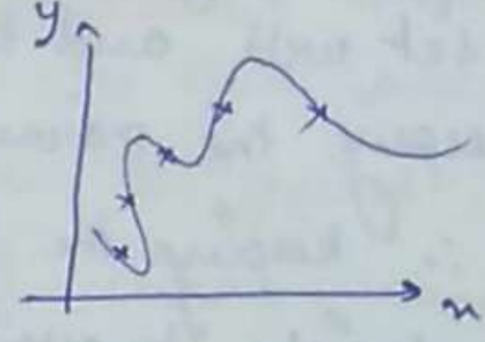


"Underfit" "high bias"

as the model seems to have a bias or preconception that y is going to linearly vary with x and despite the data to the contrary.

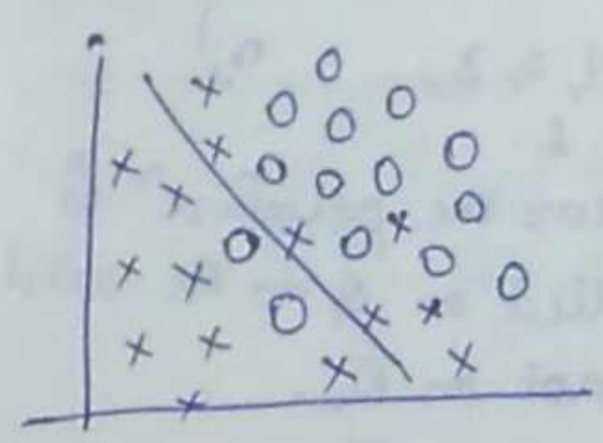


"just right"

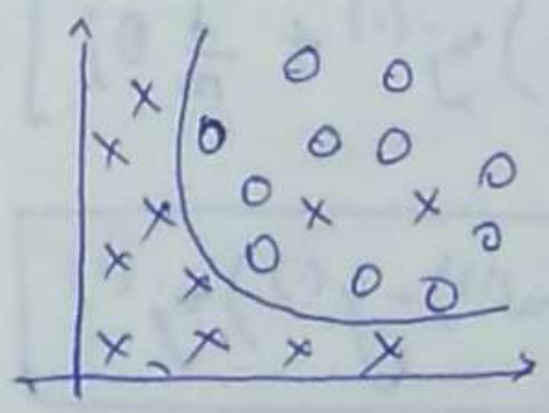


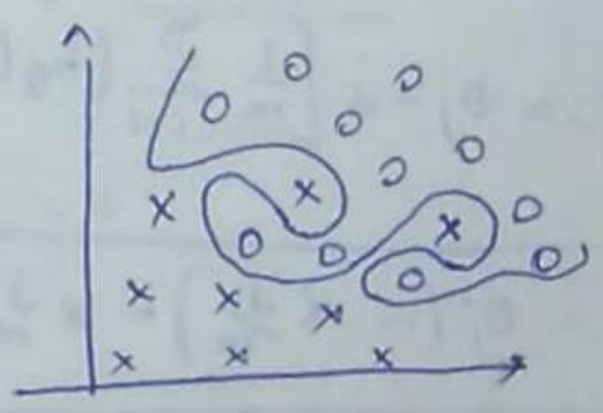
"Overfit" "high variance"

- happen when we have too many features. $h(\theta)$ fits training set very well but fails to generalize to new examples.



"Underfit"





"Overfit"

in logistic regression e.g.s.

Plotting the hypothesis is one way to decide what degree polynomial to use, but that doesn't always work as no. of parameters are usually a function of varied number of features in the dataset.

→ Address overfitting:

1. Reduce number of features (lead to little loss of info.)
   - manually select which features to keep.
   - model selection algorithm (algos which select features on their own)
2. Regularization
   - Keep all the features, but reduce magnitude / values of parameters θ_j.
   - works well when we have lot of features, each of which contributes a bit to predicting y.

for regularized $J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n}\theta_j^2\right]$

(λ: regularization parameter; $\sum_{j=1}^{n}\theta_j^2$: regularization term to shrink all the parameters θ_1 upto θ_n, as we don't know which of them to shrink in order to get simpler hypothesis which is less prone to overfitting.)

The regularization term helps in controlling the trade off b/w the 2 goals of fitting the training set well and the goal of keeping the parameter small, ∴ keeping the hypothesis relatively simple to avoid overfitting.

Note: If λ is set to an extremely large value, algorithm results in underfitting and can't even fit the training data. So, λ should be chosen with care as well. Large λ will lead to setting $\theta_1, \theta_2 \dots \theta_n \approx 0$ thus leaving $h(\theta) = \theta_0$, hence underfitting.

- Regularized linear Regression:

$$\theta_j := \theta_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \frac{\lambda}{m}\theta_j\right] \quad (j = 1, 2, 3, \dots, n)$$

↓ bcoz the penalize is applied on θ_1 to θ_n and not on θ_0.

$$\boxed{\theta_j := \theta_j\left(1 - \alpha\frac{\lambda}{m}\right) - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}}$$

$\boxed{1 - \alpha\frac{\lambda}{m} < 1}$ (to shrink the parameter a little bit) and then do the update as we used to do before.

for Normal eq^n: $\theta = (X^TX + \lambda \begin{bmatrix} 0 & & \\ & 1 & \\ & & \ddots \end{bmatrix})^{-1} X^Ty$

$(n+1) \times (n+1)$ dimensional matrix

regularize? term.

X^TX is non-invertible when $m < n$.

If $\lambda > 0$, $\theta = (X^TX + \lambda \begin{bmatrix} 0 & & \\ & 1 & \\ & & \ddots \end{bmatrix})^{-1} X^Ty$

↳ is invertible.

- Regularized logistic Regression:

$$J(\theta) = -\left[\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1-y^{(i)}) \log (1-h_\theta(x^{(i)}))\right] + \frac{1}{2m}\sum_{j=1}^{n} \theta_j^2$$

Hence, regularized can take care of overfitting even in ^nce of many features.

θ_j expression same as that for linear regression with $h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$

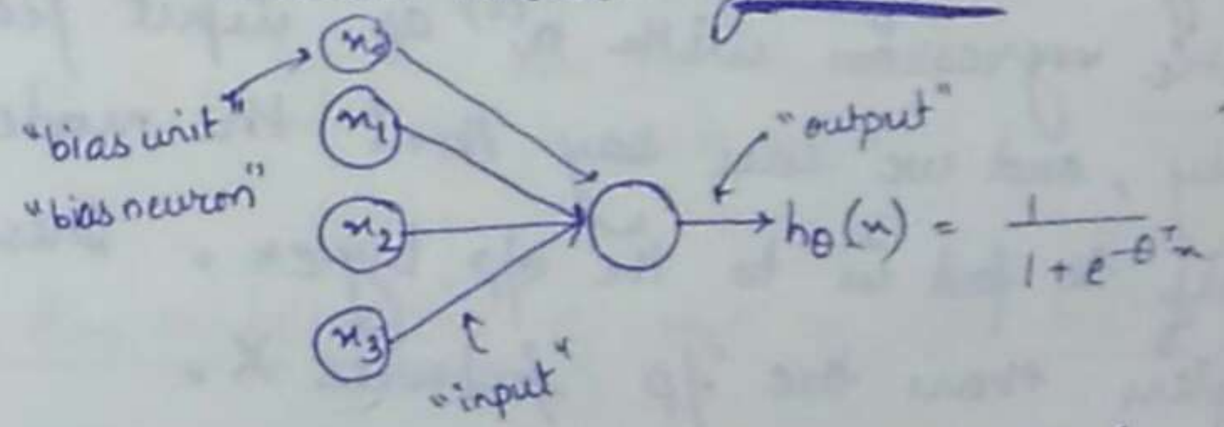
* Neural Networks: (WEEK 4)

used when we need to learn complex non-linear hypothesis, when we have a classifica^n problem with many features. In this case if we try apply logistic regression, then its difficult to guess $\theta_0, \theta_1, \ldots \theta_n$ and not only this but their square and cube terms as well and many others. for $n=100$, we will end up ~ around 5000 features to be accomodated in the equation and that too when conside only degree 2 terms.

x or the input vector for 20×20 image is a (400×3) dimensional vector to accomodate RGB code. This is too large to be handled by logistic regression.

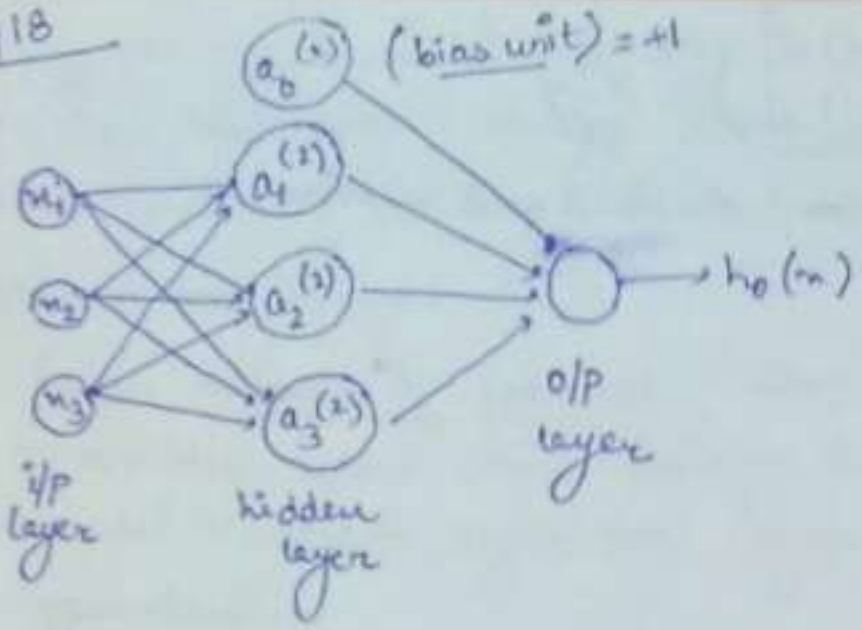
Neural networks are computationally expensive algorithms.

Neuron model: logistic unit



Artificial neuron ~ sigmoid or logistic activa^n

4/6/18



$a_i^{(j)}$ = "activa" of unit i in layer j.

↓

value that's computed by and as o/p by a specific

$\theta^{(j)}$ = matrix of weights controlling fn mapping from layer j to layer $j+1$.

$$a_1^{(2)} = g(\theta_{10}^{(1)} x_0 + \theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2 + \theta_{13}^{(1)} x_3) = g(z_1^{(2)})$$

$$a_2^{(2)} = g(\theta_{20}^{(1)} x_0 + \theta_{21}^{(1)} x_1 + \theta_{22}^{(1)} x_2 + \theta_{23}^{(1)} x_3) = g(z_2^{(2)})$$

$$a_3^{(2)} = g(\theta_{30}^{(1)} x_0 + \theta_{31}^{(1)} x_1 + \theta_{32}^{(1)} x_2 + \theta_{33}^{(1)} x_3) = g(z_3^{(2)})$$

$$h_\theta(x) = a_1^{(3)} = g(\theta_{10}^{(2)} a_0^{(2)} + \theta_{11}^{(2)} a_1^{(2)} + \theta_{12}^{(2)} a_2^{(2)} + \theta_{13}^{(2)} a_3^{(2)}) = g(z^{(3)})$$

If network has s_j units in layer j, s_{j+1} units in layer $j+1$, then $\underline{\theta^{(j)}}$ will be of dimension $s_{j+1} \times (s_j + 1)$.

Also, $a^{(1)} = x$

$z^{(2)} = \theta^{(1)} x = \theta^{(1)} a^{(1)}$

$a^{(2)} = g(z^{(2)})$

Add $a_0^{(2)} = 1$

$z^{(3)} = \theta^{(2)} a^{(2)}$

$h_\theta(x) = a^{(3)} = g(z^{(3)})$

} eqn of fwd propoga in NN

[forward propogation]

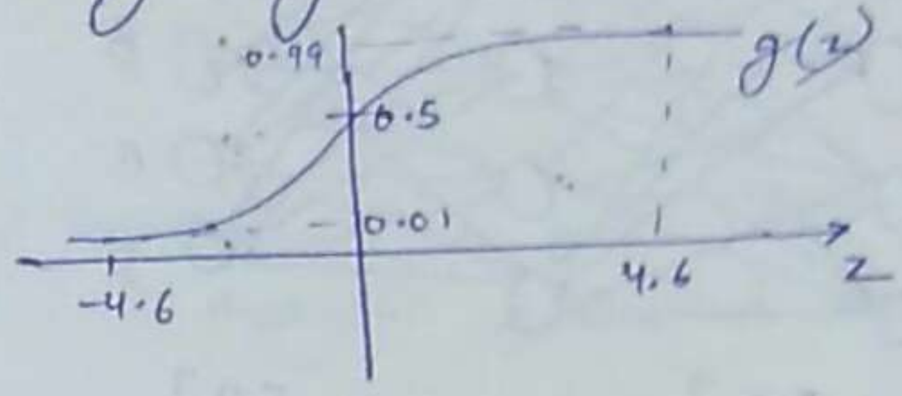
↓

bcoz we forward propogate the activation of i/p layer to calculate the activa of o/p layer

If layer 1 is removed, then the remaining part of NN is simple logistic regression with $a^{(2)}$ as input features. $a^{(2)}$ is learned from x only, and we can say that the model has learnt the features itself to feed in to the o/p layer. These learnt features are more complex than the i/p features x.

NN architecture : way in which neurons are connected to each other.

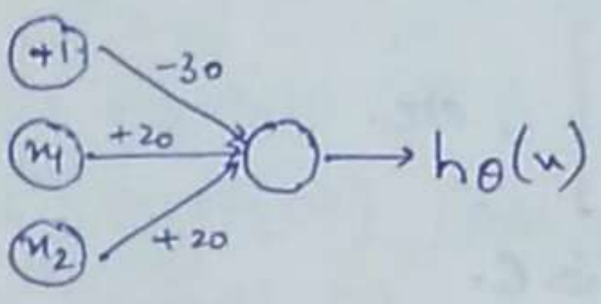
A very interesting non linear hypothesis is built by learning more & more complex features layer by layer.



e.g1 AND

$y = x_1$ AND x_2

$x_1, x_2 \in \{0, 1\}$



x_1	x_2	$h_\theta(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

$h_\theta(x) = g(-30 + 20x_1 + 20x_2)$

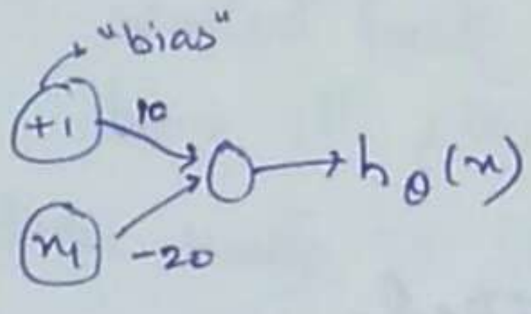
$\theta_{10}^{(1)}$ $\theta_{11}^{(1)}$ $\theta_{12}^{(1)}$

Hence $h_\theta(x) \approx x_1$ AND x_2

So, the parameter values decide which fn our NN is computing.

If $\theta_{10} = -10$, $\theta_{11} = 20$, $\theta_{12} = 20$, $h_\theta(x) \approx x_1$ OR x_2

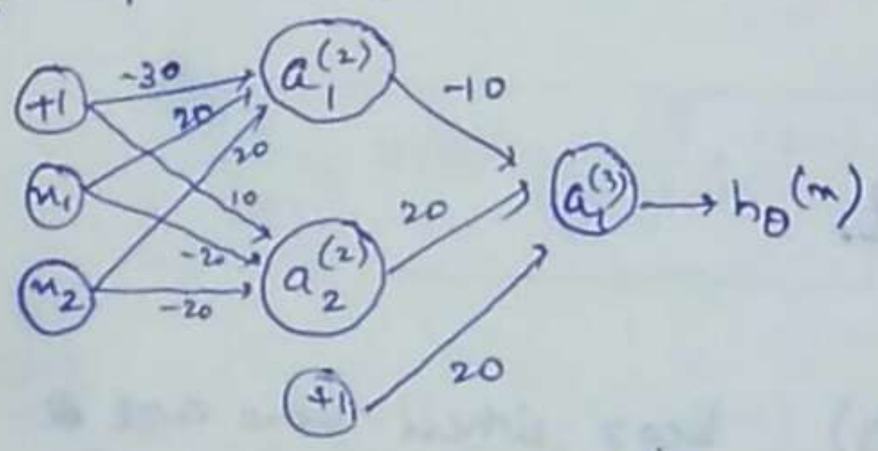
e.g2



x_1	$h_\theta(x)$
0	$g(10) \approx 1$
1	$g(-10) \approx 0$

Hence $h_\theta(x) \approx$ NOT x_1 (Negation of x_1)

e.g3 x_1 XNOR x_2



x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	$h_\theta(x)$
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

Slightly more complex fn computed by the NN as compared to x_1 and x_2.

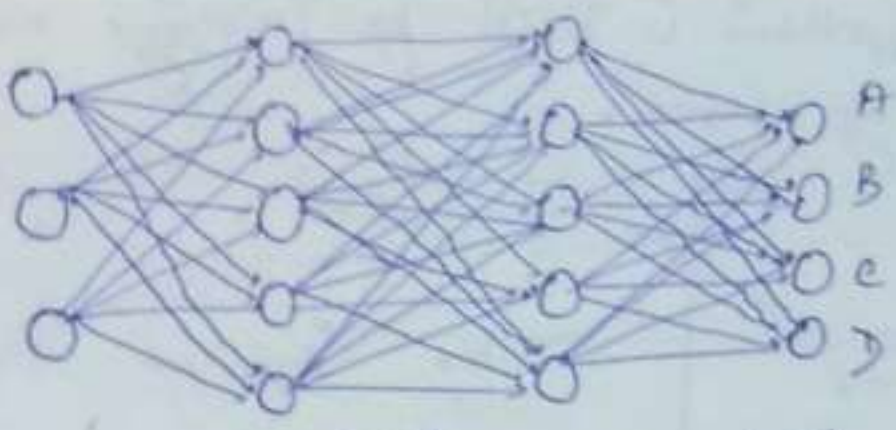
By combining the above e.g. networks, we get this network in e.g.3.

Multiclass classification in NN is an extension of one vs all technique.

e.g. to classify if the category is A, B, C or D.

MLP : Multi layer perptron

(in MLP, all layers are fully connected)



$h_\theta(x) \in \mathbb{R}^4$ ——— (1)

we want $h_\theta(x) \approx \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$, $h_\theta(x) \approx \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $h_\theta(x) \approx \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, etc.

when it's A when it's B when it's C.

where training set : $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots (x^{(m)}, y^{(m)})$

$y^{(i)}$ one of $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$ depending on if it's A, B, C or D.

$x^{(i)}$ is the image that is to be classified as A, B, C or D

$h_\theta(x^{(i)}) \approx y^{(i)}$ where both are $\in \mathbb{R}^4$.

Note $\boxed{a_1^{(o/p)} + a_2^{(o/p)} + a_3^{(o/p)} + \ldots + a_{P_i}^{(o/p)} \neq 1}$ (always)

↳ no. of classes to be identified.

the sum of activation fn need not be 1 becoz o/p of NN are not probabilities.

L = total no. of layers in network,

s_l = no. of units (not counting bias unit) in layer l.

Binary classification : $y = 0$ or 1. $s_L = 1$

Multiclass " : $y = K$ possible values $s_L = K$ $(K \geq 3)$ becoz when there are 2 classes, just 1 output layer may work.

for neural network, $J(\theta)$ is generalized of the logistic regression :-

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(h_\theta(x^{(i)})\right)_k + (1-y_k^{(i)}) \log\left(1-(h_\theta(x^{(i)}))_k\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\theta_{ji}^{(l)})^2$$

where $(h_\theta(x))_i$ = i^{th} output and $h_\theta(x) \in \mathbb{R}^K$

Backpropogation algorithm tries to minimize this cost function.

to minimize $J(\theta)$, we need code to compute $J(\theta)$ and $\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta)$ where $\theta_{ij}^{(l)} \in \mathbb{R}$ are parameters.

Let's use figure ① for one training example (x, y) and write eq's & find propoga through vectorized implementation:

$a^{(1)} = x$

$z^{(2)} = \theta^{(1)} a^{(1)}$

$a^{(2)} = g(z^{(2)})$ (add $a_0^{(2)}$)

$z^{(3)} = \theta^{(2)} a^{(2)}$

$a^{(3)} = g(z^{(3)})$ (add $a_0^{(3)}$)

$z^{(4)} = \theta^{(3)} a^{(3)}$

$a^{(4)} = h_\theta(x) = g(z^{(4)})$

Gradient computa: back propoga algo

$\delta_j^{(l)}$ = "error" of node j in layer l.

for each o/p unit (layer $L = 4$)

$$\boxed{\delta_j^{(4)} = a_j^{(4)} - y_j} \Rightarrow \underbrace{\delta^{(4)} = a^{(4)} - y}$$

vectors where each has dimension equal to no. of o/p units in our network.

$\delta^{(3)} = (\theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$

↳ element wise multiplica

where $g'(z^{(3)}) = a^{(3)} .* (1 - a^{(3)})$

$\delta^{(2)} = (\theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$

No $\delta^{(1)}$ term as 1st layer is i/p layer.

called back propoga bcoz 1st $\delta^{(4)}$ is calculated, then using it $\delta^{(3)}$ is ... and so on. bcoz we are proceeding from back, it is called so.

$$\boxed{\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) = a_j^{(l)} \delta_i^{(l+1)}}$$ Ignoring λ term of regularize

2/6/18

$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ {overall error for all training examples}

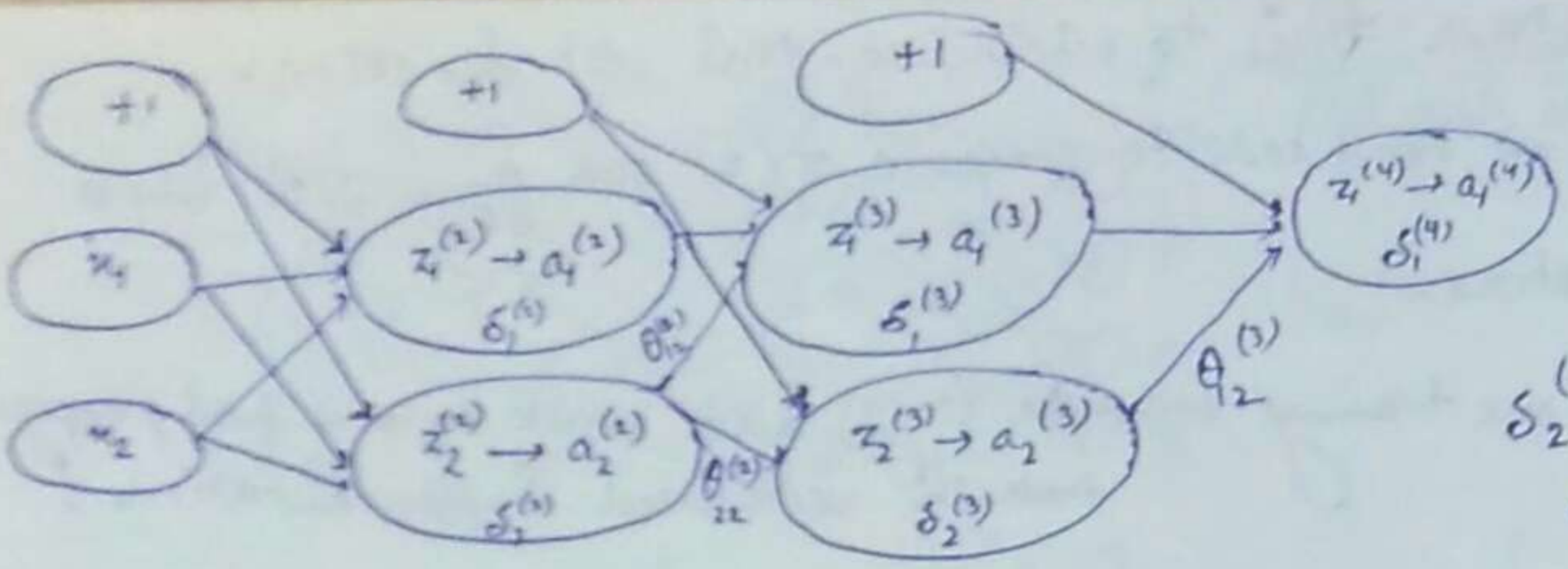
$$\boxed{\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T}$$ (vectorized implementation)

$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \theta_{ij}^{(l)}$ if $j \neq 0$

$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)}$ if $j = 0$

and $\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) = D_{ij}^{(l)}$

Backpropoga is somewhat mathematically a black box only.

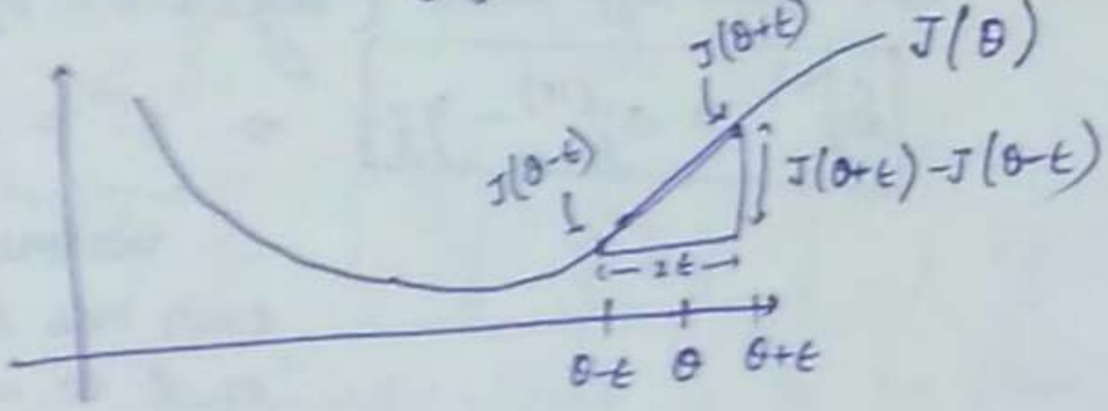


$$\delta_2^{(3)} = \theta_{12}^{(3)} \cdot \delta_1^{(4)}$$

and similarly the others.

- Gradient checking: to make sure that the implementation of fwd prop and back prop is 100% correct.

Numerical estimation of gradients -



$$\frac{d}{d\theta} J(\theta) \approx \frac{J(\theta+\epsilon) - J(\theta-\epsilon)}{2\epsilon}$$

If θ is parameter vector, then $\frac{\partial}{\partial \theta_i} J(\theta) \approx \frac{J(\theta_i + \epsilon, \theta_1 \cdots \theta_n) - J(\theta_i - \epsilon, \theta_1 \cdots \theta_n)}{2\epsilon}$

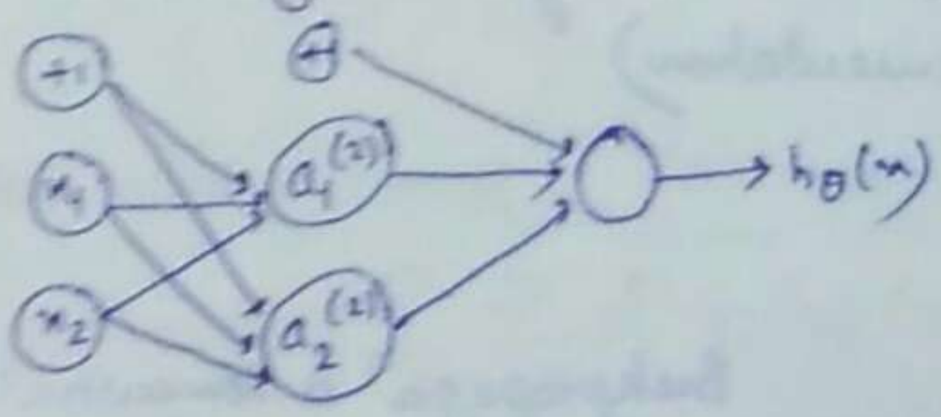
Imp! Check if $\frac{d}{d\theta} J(\theta) = DVec$ ↳ as calculated from backprop.

(unrolled $D^{(1)}$, $D^{(2)}$, $D^{(3)}$)

This gradient checking needs to be done once and then should be disabled before training the classifier bcoz if gradient check is run on every iteration of gradient descent, code will be very slow.

Initializing θ as zeros(n,1) in NN doesn't work as it works in case of logistic regression.

In case of zero initialization:



we find that if $\theta_{ij}^{(l)} = 0$ initially $\forall i,j,l$ then;

$a_1^{(2)} = a_2^{(2)}$. Also $\delta_1^{(2)} = \delta_2^{(2)}$

$$\frac{\partial}{\partial \theta_{01}^{(1)}} J(\theta) = \frac{\partial}{\partial \theta_{02}^{(1)}} J(\theta)$$

This way everytime for every iteration the weights coming from (+) will remain the same, weights coming from (x_1) will remain the same and so will those coming from (x_2), i.e all the hidden units are computing exactly the same features leading to a highly redundant representation.

Hence, we initialize the parameters of NN with random nos, this random initialize is also called symmetry breaking.

Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$
↳ different from the ϵ previously introduced.

3/6/18

No. of i/p units : dimension of features $n^{(i)}$
No. of o/p units : No. of classes
Reasonable default : 1 hidden layer, and if more than 1 then same no. of hidden units in every layer. More the hidden units, better it is.

- Training a neural network:

1) Randomly initialize weights.
2) Implement fwd propga to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$. {i.e o/p for any given ip}
3) Implement code to compute cost fn $J(\Theta)$
4) Implement backprop to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$

A for loop is used to implement backprop on all examples atleast for the first time of usage. Other alternatives for 'for' loop are complex.

5) Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed using backprop vs using numerical estimate of gradient of $J(\Theta)$. Then disable gradient checking code as its computationally very slow.
6) Use gradient descent or advanced optimaza method with backprop to try to minimize $J(\Theta)$ as a fn of parameters Θ.

Note for NN, $J(\Theta)$ is non convex and optimaza algos get stuck at local minima but it is found that these algos generally find the best local minima, if not the global minimum.

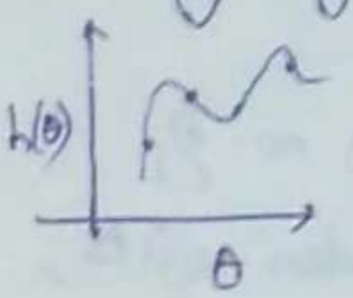
* WEEK 6 :

Suppose, an algo applied doesn't give good results on test set, then;

1) Get more training examples or
2) Try smaller set of features (to avoid overfitting) or
3) Try getting additional features (i.e. underfitting) or
4) Try adding polynomial features. ($x_1^2, x_2^2, x_1 x_2$, etc)
5) Try $\downarrow \lambda_{ij}$ or $\uparrow \lambda_{ij}$.

There are easy techniques to select which of the above options one should take to improve the model, they are ML diagnostics.

Evaluating a hypothesis i.e evaluating the hypothesis learned by the learning algorithm.



Plotting a hypothesis and then analyzing it is possible only when its in 2D but when we have many parameters, this evaluⁿ technique fails.

So, what we do is split the data into train and test set. We then learn the parameters from the training set and then calculate $J_{test}(\theta)$.

Misclassifica^n error (0/1 misclassifica^n error) : $err(h_\theta(x), y) = \begin{cases} 1 & \text{if } h_\theta(x) \geq 0.5,\ y = 0 \\ & \text{or if } h_\theta(x) < 0.5,\ y = 1 \\ 0 & \text{otherwise} \end{cases}$ } error

$$\text{Test error} = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} err\left(h_\theta(x_{test}^{(i)}), y^{(i)}\right)$$

this denotes "fraction" of the test set examples that have been misleveled by our hypothesis.

Model selⁿ problems : deciding value of λ, deciding which features to keep or which to reject.

- Training set error is not a good indicator of how well the hypothesis will do on new examples, bcz the training set error is likely to be lower than the actual generalizⁿ error.

- Suppose we want to decide which degree of polynomial should be selected (d). For this, we find $J_{test}(\theta)$ for different values of d and select that value of d which gives minimum $J_{test}(\theta)$. Hence, this is fitting the parameter 'd' on test data. So, there is less chances that this error is similar to the generalized error i.e with the examples that the model has not seen before. Hence, to solve this issue, we divide our data into train set, valida^n/cross valida^n set, test set. where approx 60% is train, 20% is valida^n & 20% is test data.

Now,

training error : $J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$

Cross valida^n error : $J_{cv}(\theta) = \frac{1}{2m} \sum_{i=1}^{m_{cv}} (h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$

Test error : $J_{test}(\theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} (h_\theta(x_{test}^{(i)}) - y_{test}^{(i)})^2$

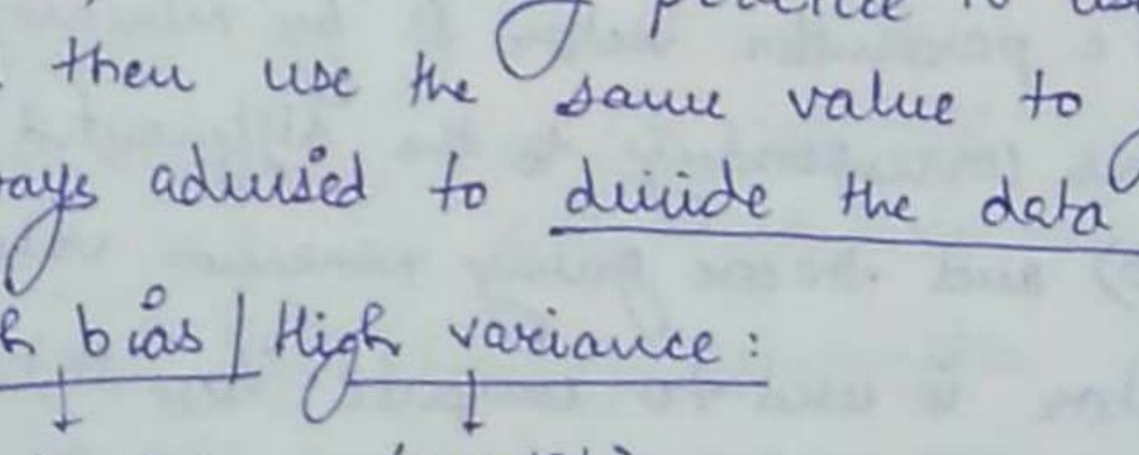
So, now instead of ~~selected~~ using test set for model selec^n, we will use cv set for model selection. Now the d is fitted on cv set and so now test set can be used to estimate the generaliza^n error of the model that was selected.

Hence, it is a wrong practice to use the test set to select the model and then use the same value to generalize the model. It is always advised to divide the data into 3 parts.

- High bias / High variance:

(underfit) (overfit)

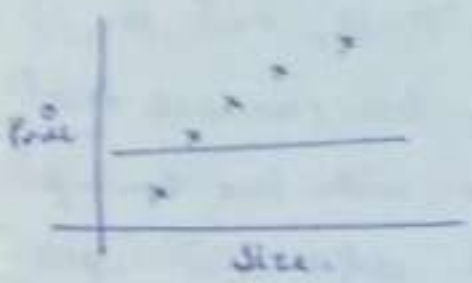
Q Suppose our learning algo is performing less well than we were hoping. (i.e $J_{cv}(\theta)$ or $J_{test}(\theta)$ is high). Is it a bias or a variance problem?



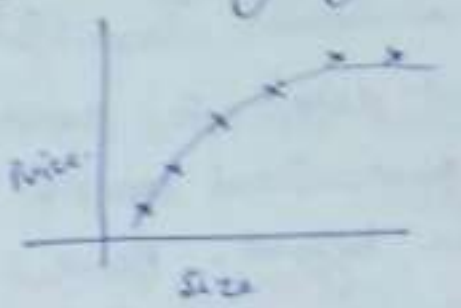
Bias (underfit) : $J_{train}(\theta)$ as well as $J_{cv}(\theta)$ will be high.

Variance (overfit): $J_{train}(\theta)$ is low while $J_{cv}(\theta)$ will be high.

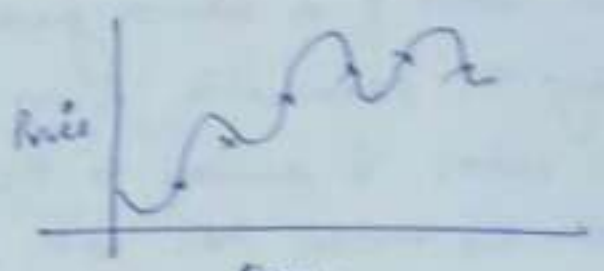
Regularization helps prevent overfitting but how does it affect the bias & variance of a learning algorithm.



large λ
High bias (underfit)
$\lambda = 10000$, $\theta_1 \approx 0$, $\theta_2 \approx 0$, ..
$h_\theta(x) \approx \theta_0$



Intermediate λ
"Just right"



Small λ
High variance (overfit)
$\rightarrow \lambda = 0$

given: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

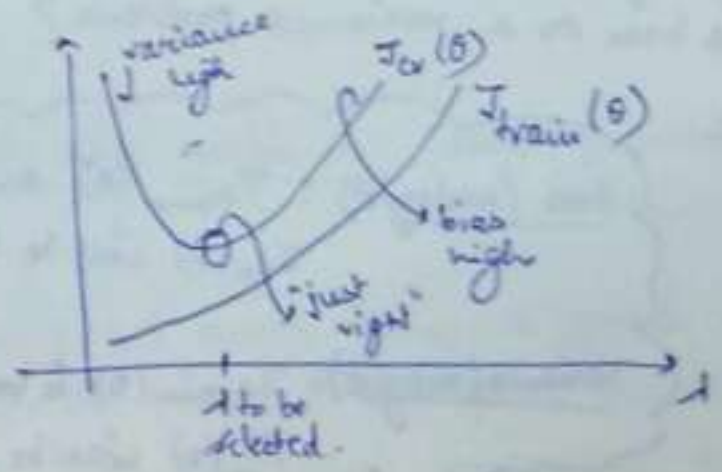
$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + \underbrace{\frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2}_{\text{regularized term}}$$

Choosing the regularization parameter 'λ':

1. Try $\lambda = 0$
2. " $\lambda = 0.01$
3. " $\lambda = 0.02$
:
12. Try $\lambda = 10$

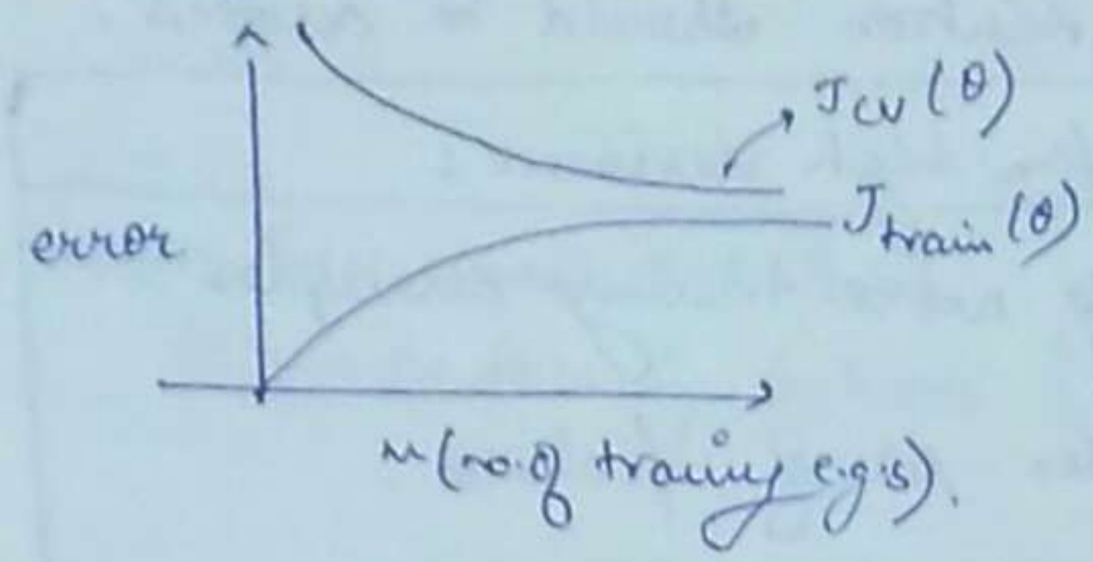
} Similarly trying out a set of values of λ gives 12 different models

Now, every λ will correspond to a parameter vector θ by minimizing $J(\theta)$. Now these parameter vectors corresponding to the different λ values were used to compute $J_{cv}(\theta)$ and the one giving minimum value for $J_{cv}(\theta)$ is selected. Now this θ vector is used to compute $J_{test}(\theta)$ to report the error. So, that's model selection applied to selecting the regularization parameter 'λ'.

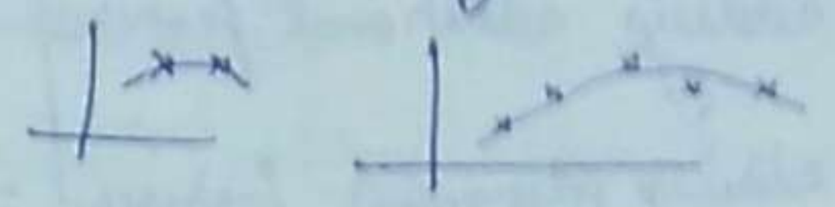


Hence, some intermediate value of λ will work best in terms of having small $J_{cv}(\theta)$ or small $J_{test}(\theta)$.

→ Learning curves : to check and improve the performance of algos.



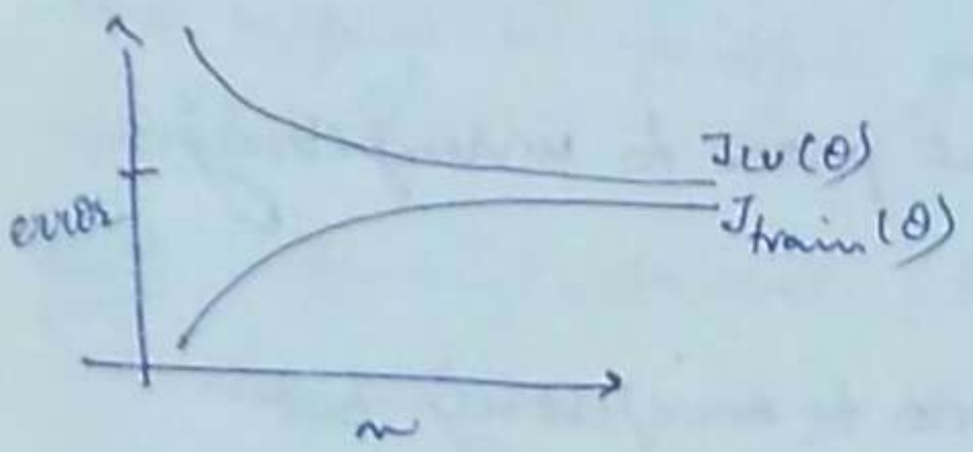
this is bcoz when we have less training e.g. it is easier to fit them in an eg:



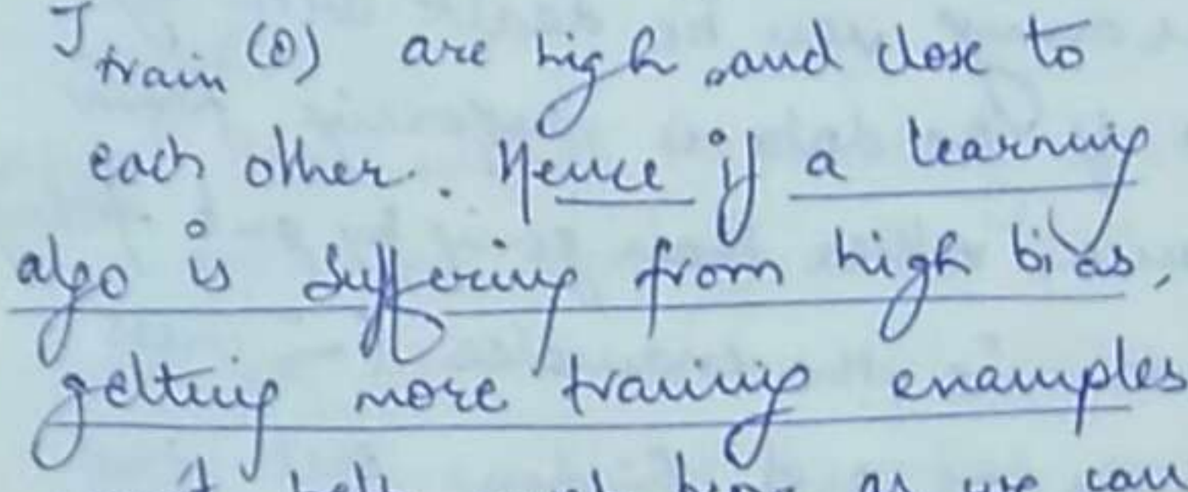
but when m is large, it gets harder to fit the hypotheses to them.

$J_{cv}(\theta)$ tes with ↑ing m bcoz with greater m, a more generalised hypothesis can be made.

High bias case

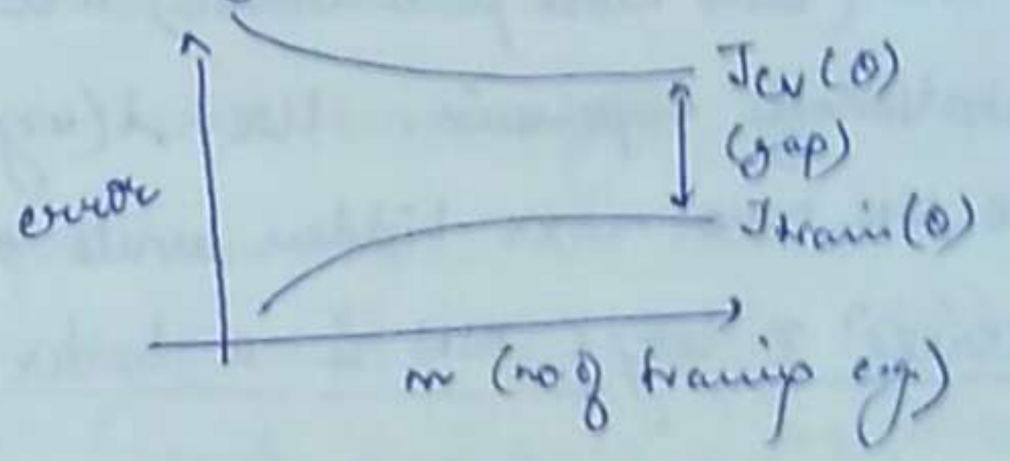


In case of high bias, performance of the hypothesis will be quite similar on the training set as well as the testing i.e cross validation set. In this case both $J_{cv}(\theta)$ and $J_{train}(\theta)$ are high and close to each other. Hence if a learning algo is suffering from high bias, getting more training examples won't help much bcoz as we can see $J_{cv}(\theta)$ is not ↓ing much with ↑ing m, it is only getting flattened bcoz in case of high bias:

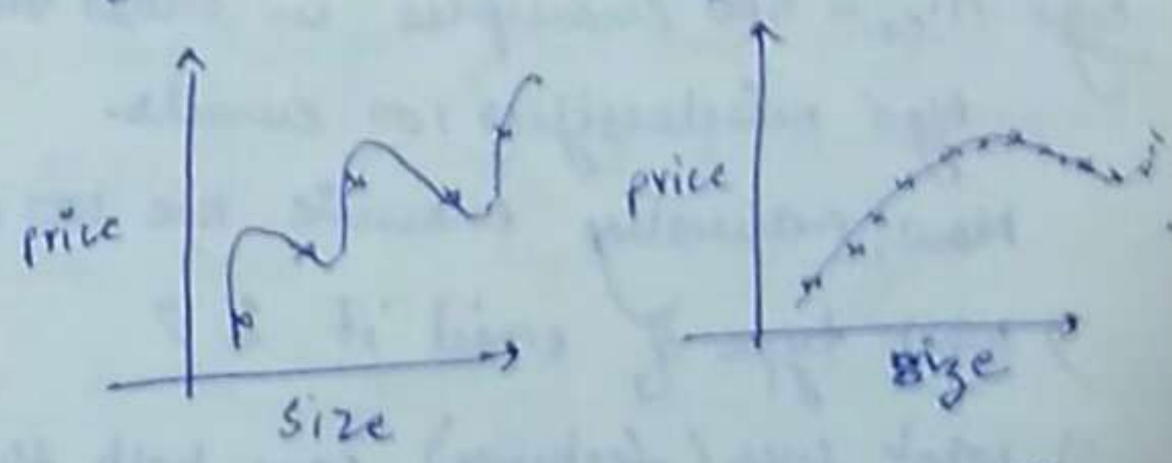


$h_\theta(x) = \theta_0 + \theta_1 x$
(large λ)

High variance case.



In case of high variance, getting more training data helps, bcoz $J_{cv}(\theta)$ and $J_{train}(\theta)$ seems to be converging towards each other with ↑ing m. This is bcoz in case of high variance $h(\theta)$ seems to less overfit the data in case of greater no. of training examples.



$h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_{100} x^{100}$
(small λ)

Hence, drawing these learning curves help us learn if there is a bias or a variance problem and hence what solution should be adopted.

★ So to fix high bias:	★ to fix high variance:
- Try adding additional features - Try adding polynomial features - Try ↓ in λ.	- Try getting more training examples. - Try smaller set of features. - Try ↑ in λ.

→ Neural networks and overfitting:

- Small NN (having fewer parameters) are more prone to underfitting but computationally cheaper.
- Large NN (with more parameters) are more prone to overfitting but computationally expensive. Use λ (regularize) to address overfitting. Large NN have more hidden units or more hidden layers. Using regularized & large NN is a better choice than using small NN.

→ Error Analysis: (to be done on CV set & not test set)

Premature optimize in computer programming can be dealt with by help of plotting learning curves to know if the data is differing from high bias or high variance in advance, rather than going by gut feeling.

Error analysis refers to manually examine the examples (in cross validation set) that our algo made errors on and finding out the systematic trend if any of examples the algo makes errors on.

e.g. $m_{cv} = 500$ examples in cross validation set

Algo misclassifies 100 emails.

Now, manually examine the 100 errors, and categorize them based on:

1) what type of email it is?

2) what cues (features) can help the algo classify them correctly.

Suppose, emails are found to be ← Pharma : 12
Replica/fake : 4
intended to steal password : 53 ←
other : 31.

Suppose these emails are found to have the following features ← Deliberate misspellings : 5
Unusual email routing : 16
Unusual use of punctuations : 32 ←

Now, an algo can be designed that focuses on identifying the unusual punctuation use and help in better identifying the spam mails, thus helping in fruitful guidance towards making a successful model.

Note "Stemming software (Porter stemmer)" helps treating discount/ discounts/ discounting/ discounted as same words. But it might hurt when it considers universe and university to be same.

To check if stemming is doing well for our algo, we can calculate the cv error with and without stemming and then easily decide whether to involve stemming or not, and similarly for other kind of ideas as well, bcoz checking manually everytime would be tough.

Note This initial exercise of error analysis and all is a powerful tool for deciding where to spend the time next bcoz first we look at the errors it makes and do error analysis to see what other mistakes it makes and use that to inspire further development.

Skewed classes : when examples of 1 class highly outnumber the examples of other class. In such cases using just one single no. evaluation doesn't work properly. e.g. if there is a dataset with only 0.5% patients having cancer and I have an algo which always predicts the patient to be healthy, then this single no. evaluation gives that algo is 99.5% correct which is actually not the case.

Hence in case of skewed classes, this single value evaluation fails. So, we come up with Precision / Recall evaluation metric.

Predicted class \ Actual class	1	0
1	True +ve	False +ve
0	False −ve	True −ve

$y=1$ in case of rare class that we want to detect.

Precision: Of all the patients where we predicted $y=1$, what fraction actually has cancer?

$$\frac{\text{True positives}}{\#\text{ predicted +ve}} = \boxed{\frac{\text{True +ve}}{\text{True +ve + False +ve}}}$$

* High precision is good.

sensitivity: true +ve rate
specificity: true −ve rate

Recall / Or Sensitivity: Of all the patients that actually have cancer, what fraction did we correctly detect as cancer?

$$\boxed{\frac{\text{True positives}}{\text{True +ve + False −ve}}} = \frac{\text{True positives}}{\#\text{ actual positives}}$$

* High recall is good and specificity $= \boxed{\frac{\text{True negatives}}{\#\text{ actual negative}}}$

So, precision/recall will give a better sense of how well our classifier is working. A classifier with high precision and high recall is actually a good classifier as it cleans the whole scene and can't cheat by predicting $y=0$ or $y=1$ all the time. Bcoz if it predicts $y=0$ all the time, then precision ↑ but recall is low, i.e a bad classifier. If $y=1$ all the time then high recall but low precision.

→ Trading off precision & recall: (using the threshold value)

Suppose we want to predict $y=1$ (cancer) only if very confident.

One way to do that is to modify the hypothesis $h_\theta(x)$ from

$h_\theta(x) \geq 0.5$ Predict 1
$h_\theta(x) < 0.5$ Predict 0

to

$h_\theta(x) \geq 0.7$ Predict 1
$h_\theta(x) < 0.7$ Predict 0

i.e

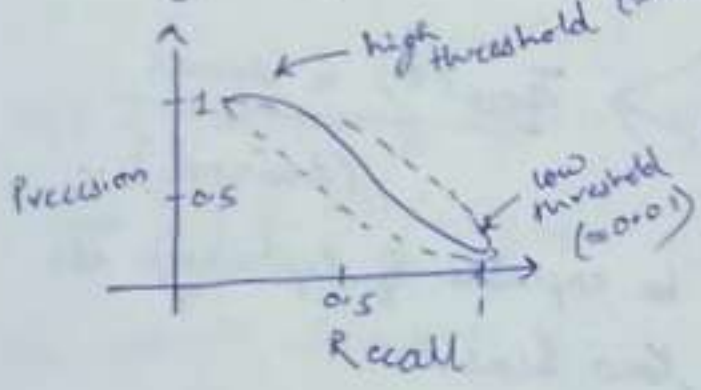
we say patient has cancer only if he has 70% or more chance of having cancer, leading to <u>higher precision</u> but in contrast this classifier will have <u>lower recall</u> bcoz now $y=1$ is predicted for a smaller no. of patients.

But, suppose we want to avoid missing too many cases of cancer (i.e avoid false -ves) bcoz suppose a patient has cancer & we fail to tell it to them, it can be really bad as then they might not get the desired treatment. So, in case of doubt also we want to predict $y=1$, in this case we change $h_\theta(x)$ to

$h_\theta(x) \geq 0.3$, Predict 1
$h_\theta(x) < 0.3$, Predict 0

In this case, we have <u>higher recall</u> but <u>lower precision</u>.

Hence, it depends from case to case if we need higher recall or higher precision.



where $h_\theta(x) \geq$ threshold.

this precision - recall graph can take many shapes depending on the classifier details.

<u>Q.</u> How to decide this threshold automatically and how to decide which algo is better like in case below:

	Precision	Recall	Avg.
Algo 1	0.5	0.4	0.45
Algo 2	0.7	0.1	0.4
Algo 3	0.02	1.0	0.51

** In case of single no. evaluation this problem didn't arise but now we have two nos. to decide upon.

To convert this precision-recall to a single value evaluation metric: we can calculate $\boxed{\text{Average} = \frac{\text{Precision} + \text{Recall}}{2}}$ and the one with higher avg. is the better algorithm. But this always doesn't work, well as seen in previous example. Algo 3 has such low precision but still is assumed to be best based on avg. value.

So, we come up with the reality out of precision & recall

$$\boxed{\text{F Score or } F_1 \text{ Score} = \frac{2PR}{P+R}}$$

If either P or R is zero FScore is 0, so for a good F Score both P and R need to be reasonably high. Perfect F score = 1 when $P = R = 1$.

Q. How much data to train on?

"It's not who has the best algo that wins, but who has the most data".

Note. Using a learning algo with many parameters helps having low bias and then using a large dataset helps having low variance.

* low bias $\Rightarrow J_{train}(\theta)$ is small.
  low variance $\Rightarrow J_{train}(\theta) \approx J_{test}(\theta)$
  $\Rightarrow J_{test}(\theta)$ is small (desired)

Note parameters of the learning algo must be capable of capturing all the features of the dataset to have low bias.

Hence, large such parameters with large dataset give high accuracy algos.

→ This can be done by comparing it with human, i.e given features x, can a human successfully predict y. If yes, that means the features are or the parameters are sufficient for good accuracy algo.

Note: [logistic and linear regression along z NN are low bias algos]

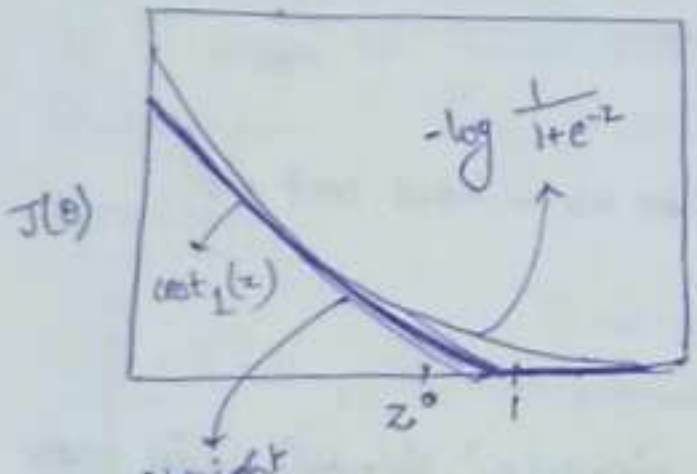
* WEEK 7:

4/6/18

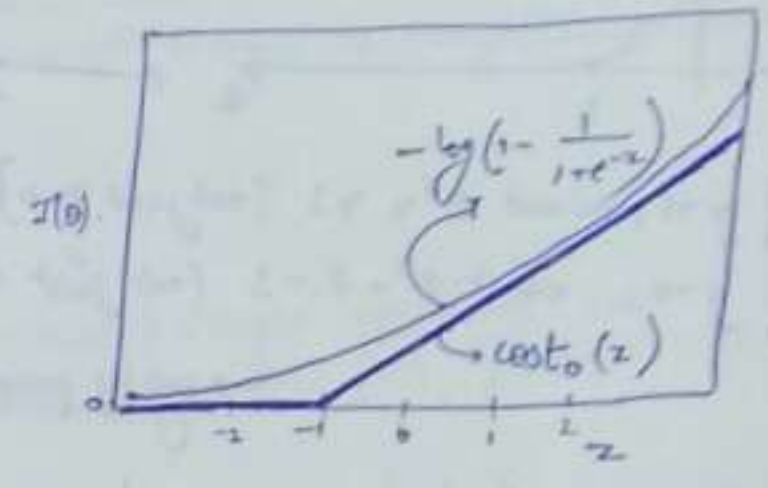
⇒ Support Vector Machines: (Large Margin Classifier)

This algo gives a more cleaner and powerful way of learning complex linear functions. It can be obtained by modifying logistic regression a bit.

If $y=1$ (want $\theta^T x$ i.e $z \gg 0$):



If $y=0$ (want $\theta^T x \ll 0$):



$cost_1(z)$: cost corresponding to $y=1$.

$cost_0(z)$: " " " " $y=0$.

Cost fn in case of logistic regression was:

$$\min_\theta \frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)}\left(-\log h_\theta(x^{(i)})\right) + \left(1-y^{(i)}\right)\left(\left(-\log\left(1-h_\theta(x^{(i)})\right)\right)\right]\right] + \frac{1}{2m}\sum_{j=1}^{n}\theta_j^2$$

So, cost fn for SVM is:

$$\min_\theta \frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)}\, cost_1(\theta^T x^{(i)}) + \left(1-y^{(i)}\right) cost_0(\theta^T x^{(i)})\right] + \frac{1}{2m}\sum_{i=0}^{n}\theta_j^2 \quad -①$$

$$\equiv \boxed{\min_\theta C\sum_{i=1}^{m}\left[y^{(i)}\, cost_1(\theta^T x^{(i)}) + \left(1-y^{(i)}\right) cost_0(\theta^T x^{(i)})\right] + \frac{1}{2}\sum_{i=1}^{n}\theta_j^2} \quad -② \quad \text{(for SVM)}$$

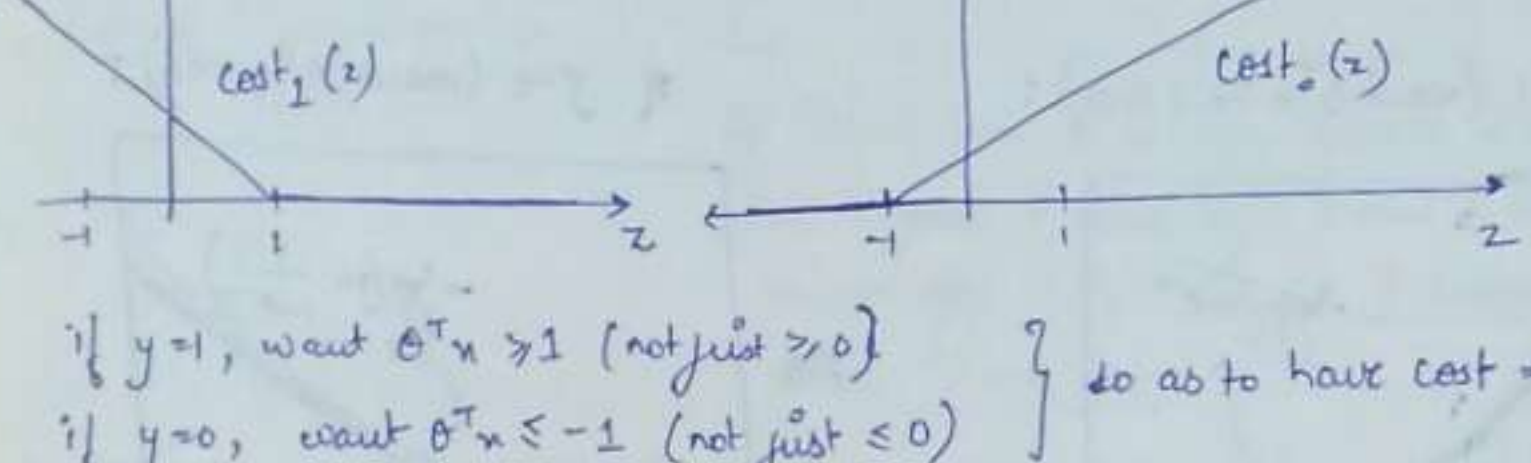
Instead of notation $A+\lambda B$, it is written as $CA+B$ where if C is less B is given more weight just as when λ was more B was given more (work like $\frac{1}{\lambda}$)

C is also regularization parameter

weight. Also m is removed bcoz it was in multiplica^ and removing it does not changed the desired minimum value of θ.

If we put $C = \frac{1}{\lambda}$, the cost fN ① and ② turn out to be same.

So, $h_\theta(x)$ for SVM is $\begin{cases} 1 & \text{if } \theta^T x \geq 0 \\ 0 & \text{otherwise} \end{cases}$ unlike that of logistic regression where $y=1$ if $h_\theta(x) \geq 0.5$ and $y=0$ otherwise.



if $y=1$, want $\theta^T x \geq 1$ (not just ≥ 0)
if $y=0$, want $\theta^T x \leq -1$ (not just ≤ 0)
} do as to have cost = 0.

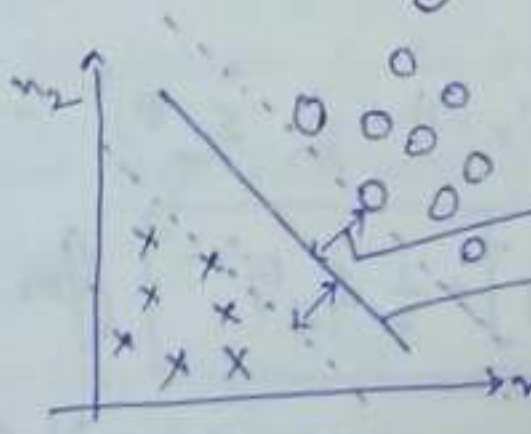
↳ safety purpose.

If C is chosen to be very very large then the minimize^ objective will chose a value so that $A = 0$.

$\Rightarrow \min_\theta \frac{1}{2} \sum_{i=1} \theta_j^2$ such that $\theta^T x^{(i)} \geq 1$ if $y^{(i)} = 1$ and $\theta^T x^{(i)} \leq -1$ if $y^{(i)} = 0$.

On solving this optimize^ problem, when minimizing this as fN of parameter θ, we get a very interesting decision boundary.

e.g. SVM decision boundary : linearly separable case



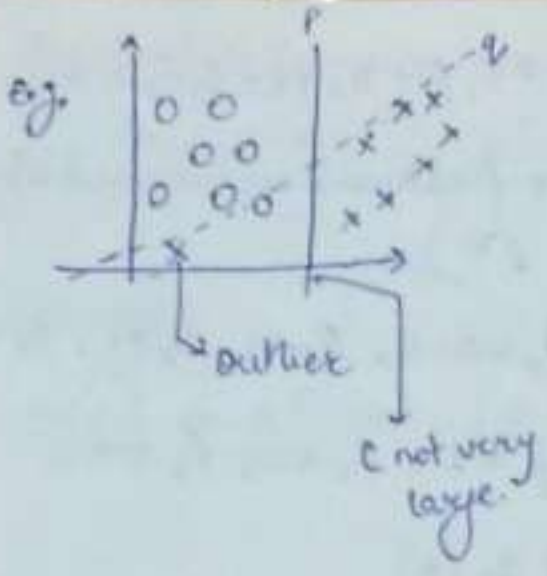
'margin' of SVM

decision boundary (hyper plane)

Selected by SVM having some larger minimum distance from any of the training examples.

Note

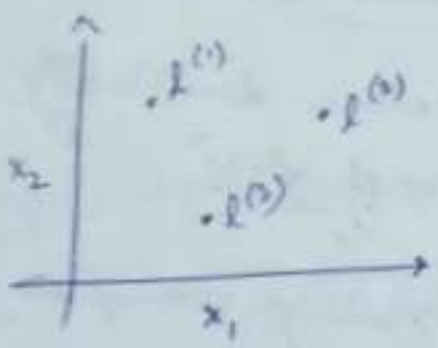
SVM tries to separate the data with as large margin as possible.



Using a large margin classifier may be quite sensitive to outliers. If C is very large (i.e $\frac{1}{\lambda}$ is small) 'q' boundary will be selected which is actually not a good choice to make just beoz of one outlier. This decision boundary is actually a sign of overfitting. 'p' is the right decision boundary and it will be selected when the regularize parameter C is reasonably small i.e λ is large (avoiding overfit).

Even if the data is not linearly separable, SVM does its work well. It is better to ignore outliers than to change to overfitting decision boundary.

- Kernels :



Given x, compute new features depending on proximity to landmarks $l^{(1)}, l^{(2)}, l^{(3)}$.

Given x: $f_1 = \text{similarity}(x, l^{(1)})$

$= \exp\left(-\frac{\|x - l^{(1)}\|^2}{2\epsilon^2}\right)$ where $\|x - l^{(1)}\|$ denotes Euclidian distance b/w the two.

$f_2 = \text{similarity}(x, l^{(2)})$

$f_3 = \text{similarity}(x, l^{(3)}) = K(x, l^{(3)})$

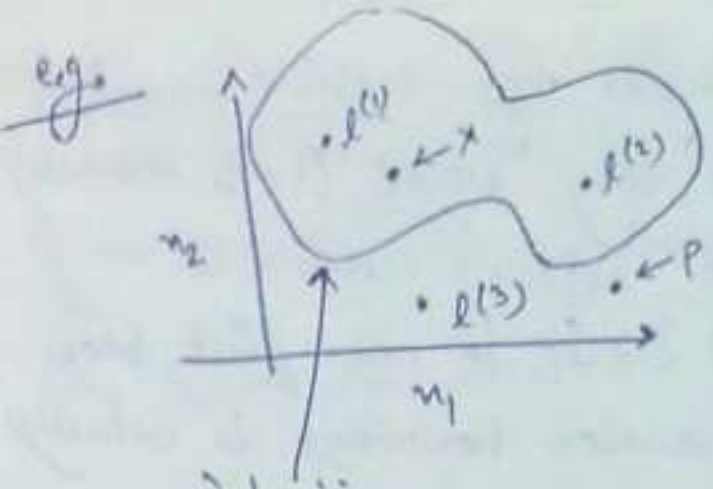
↑ "Kernel function" (Gaussian Kernel)

If $x \approx l^{(i)}$: $f_1 \approx \exp\left(-\frac{0^2}{2\sigma^2}\right) \approx 1$

If x is far from $l^{(i)}$: $f_1 \approx \exp\left(-\frac{(\text{large number})^2}{2\sigma^2}\right) \approx 0$

Each of the landmarks define a new feature, i.e a given training e.g. we can compute three features f_1, f_2, f_3 given $l^{(1)}, l^{(2)}, l^{(3)}$.

e.g.



decision boundary
(inside $y=1$ and outside $y=0$.)

Predict 1 when $\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 \geq 0$

(where f_1, f_2, f_3 are features corresponding to $l^{(1)}$, $l^{(2)}$ and $l^{(3)}$)

Suppose we calculated $\theta_0 = -0.5$, $\theta_1 = 1$, $\theta_2 = 1$, $\theta_3 = 0$

for the training e.g. X; $f_1 \approx 1$, $f_2 \approx 0$, $f_3 \approx 0$

So, $\theta_0 + \theta_1(1) + \theta_2(0) + \theta_3(0)$

$= -0.5 + 1 = 0.5 \geq 0$

Hence, $\underline{y=1}$ is predicted for X.

Similarly for a point p, we get $h(\theta) = -0.5 \leq 0$. Hence $\underline{y=0}$ for this point.

In this e.g. we conclude that for points near $l^{(1)}$ and $l^{(2)}$, $y=1$ and for points far away from $l^{(1)}$ and $l^{(2)}$, $y=0$.

In this way, we can learn pretty non linear boundaries.

Q. How do we chose these landmarks and what other similarity functions can we use other than the Gaussian Kernel fn?

→ Given $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots (x^{(m)}, y^{(m)})$ for m training examples

choose $l^{(1)} = x^{(1)}$, $l^{(2)} = x^{(2)}$, ..., $l^{(m)} = x^{(m)}$

Given example x:

$f_1 = \text{similarity}(x, l^{(1)})$

$f_2 = \text{similarity}(x, l^{(2)})$

$f = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_m \end{bmatrix}$ $f_0 = 1$

for training example $(x^{(i)}, y^{(i)})$:

$f_1^{(i)} = \text{sim}(x^{(i)}, l^{(1)})$

$f_2^{(i)} = \text{sim}(x^{(i)}, l^{(2)})$

$\vdots$

$f_m^{(i)} = \text{sim}(x^{(i)}, l^{(m)})$

$\leftarrow$ $f_1^{(i)} = \text{sim}(x^{(i)}, l^{(i)}) = \exp\left(\frac{-0}{2\sigma^2}\right) = 1$ ($x^{(i)}$)

Hence, one of these features = 1.

Now $x^{(i)}$ can be represented as $f^{(i)} = \begin{bmatrix} f_0^{(i)} \\ \vdots \\ f_m^{(i)} \end{bmatrix}$

Hypothesis: Given x, compute features $f \in R^{n+1}$

Predict "y=1" if $\underline{\theta^T f} \geq 0$.

$\hookrightarrow = \theta_0 f_0 + \theta_1 f_1 + \theta_2 f_2 + \dots \theta_n f_n \quad \{\theta \in R^{n+1}\}$

Training:

$$\min_{\theta} C \sum_{i=1}^{m} y^{(i)} \text{cost}_1(\theta^T f^{(i)}) + (1-y^{(i)}) \text{cost}_0(\theta^T f^{(i)}) + \frac{1}{2} \sum_{j=1}^{n} \theta_j^2$$

(Summing θ upto n) — solving this we get parameters for SVM.

$\sum_j \theta_j^2 = \theta^T \theta$, $\theta = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$ (ignoring θ_0 as we do while regularize.)

instead of minimizing $\|\theta\|^2$, SVM minimizes $\theta^T M \theta$
↳ some matrix to take care of computational efficiencies, like if no. of training eg. is too large then modified helps in computation.

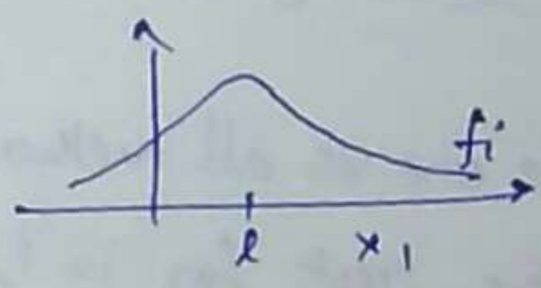
Note SVM and kernels go well together but logistic regression would run very slowly with kernels.

Q How to choose C for SVM? , σ^2 for SVM?
↓
for optimize purpose.

Ans Large C: lower bias & high variance (small λ) {prone to overfitting}
Small C: higher bias & lower variance. (large λ) {prone to underfitting}

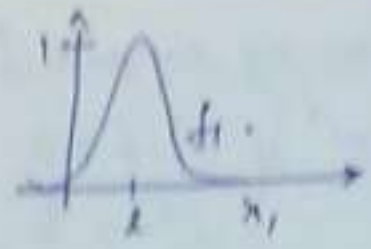
Large σ^2: Features f_i vary more smoothly
- Higher bias, lower variance. bcz Gaussian Kernel that varies smoothly tends to give a hypothesis that varies slowly and smoothly

$\left\{\exp\left(-\frac{\|x-l^{(i)}\|^2}{2\sigma^2}\right)\right\}$



Gaussian Kernel tends to fall off slowly when σ^2 is large.

Small σ^2 : features vary less smoothly
lower bias, higher variance.



Eg. No Kernel ("linear kernel")

Predict "y=1" if $\theta^T x \geq 0$

This kernel is generally chosen when n is large and m is small to avoid overfitting.

(other common)

eg. Gaussian kernel $(f_i = e\left(-\frac{\|x - l^{(i)}\|^2}{2\sigma^2}\right))$

where $l^{(i)} = x^{(i)}$.

This kernel is used when n is small but m is large to fit a more complex decision boundary.

Note Do perform feature scaling before using the Gaussian Kernel, bcoz otherwise some features with larger values would be given more weight/importance than other features with smaller values. e.g. size of house would be given more weightage than no. of bedrooms if scaling is not done.

Not all similarity fn, similarity(x, l) make valid kernels. The kernels need to satisfy technical condition called "Mercer's theorem" to make sure that SVM packages optimize run correctly to calculate the parameters.

Other kernels :

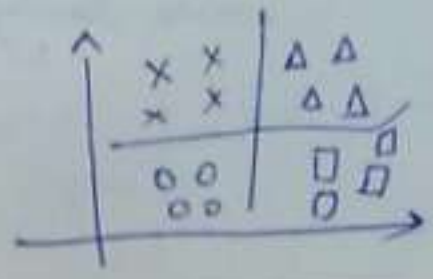
→ Polynomial kernel : $k(x, l) = (x^T l)^2$ or $(x^T l)^3$, $(x^T l + 1)^3$, etc.

The constant added and the degree of polynomial vary from 1 type of polynomial kernel to another.

→ String kernel, chi square kernel, histogram kernel, intersect kernel, ...

String kernel used when x is text string.

⇒ Multiclass classification – SVM classifiers have generally built in multiclass classification functionality. Other method is to use the one vs all method where K SVMs are trained to distinguish $y = i$ from the rest for $i = 1, 2, \ldots K$ to get $\theta^{(1)}, \theta^{(2)}, \ldots \theta^{(K)}$. Then the class with largest $(\theta^{(i)})^T x$ is picked up.



Sup when to use SVM and when to use logistic regression?

Ans n = no. of features ($x \in R^{n+1}$), m = no. of training examples.

- If n is large (relative to m) : Use LR or SVM without kernel ("linear")
  e.g. in text classification problems.

  bcoz when n is small, we don't have much data and so linear f^N would probably do fine as we need not fit a very complicated non linear function.

- If n is small, m is intermediate : Use SVM with Gaussian kernel.
  (n= 1-1000, m = 10-10k)

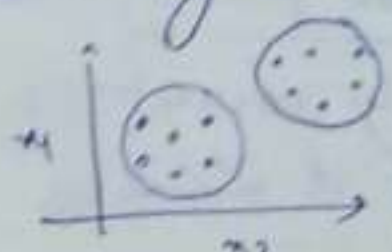
- If n is small, m is large : Create or add more features, then use LR or linear SVM, as in this case Gaussian SVM would be slow to run.
  (n=1-1000, m = 50,000+)

Note LR and linear SVM are quite similar. NN will work well for most of the above settings but they may be slower to train. The optimize problem that SVM has is a convex optimize problem and so good SVM optimize s/w packages will always find a global minimum or something close to it, so for SVM we need not worry about local optima and for NN also, local optima is not a problem as it always finds global optima.

So, SVM : a very powerful tool to learn complex non-linear f^N.
- linear SVM gives linear decision boundary. just like LR does.

WEEK 8 :

Clustering : An Unsupervised learning algorithm.



Training set: $\{x^{(1)}, x^{(2)}, x^{(3)}, \ldots x^{(m)}\}$

we need an algo to find some structure in the data for us.

An algo that finds clusters like the ones just circles is called clustering algorithm. Its applica^n areas are:

1) Market segmentation
2) Social network analysis
3) Astronomical data analysis.

→ K-Means algorithm : a specific clustering algorithm. (most popular)

5/6/18 Steps:

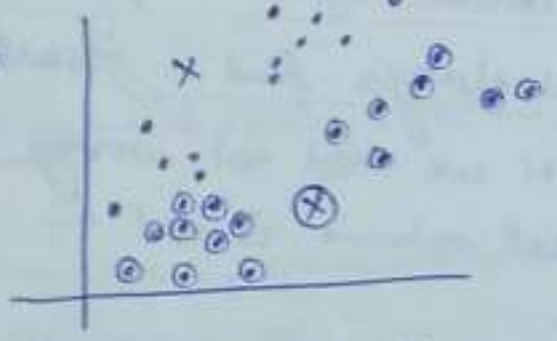
1) Randomly initialize two points called cluster centroids, when I want to group data into two clusters.

K Means is an iterative algorithm and it does two things -

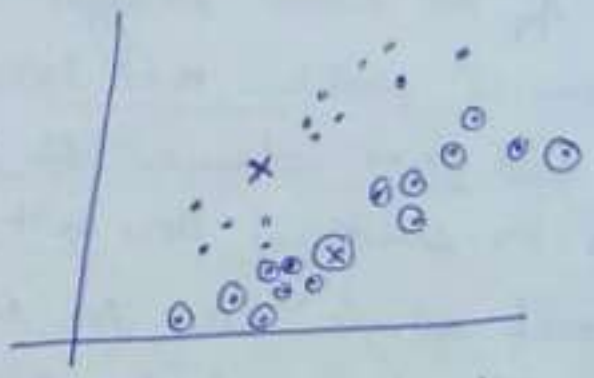
1) cluster assignment step
It checks all the datapoints and depending upon to which centroid the data point is closer, that cluster is assigned to it.

2) Move centroid step
the centroids are moved to the average of the points assigned to the same cluster.

e.g.



Step 2 →



⊗ and × are cluster centroids randomly initialized. Datapoints close to ⊗ are marked as ⊙ and datapoints close to × are marked as ·.

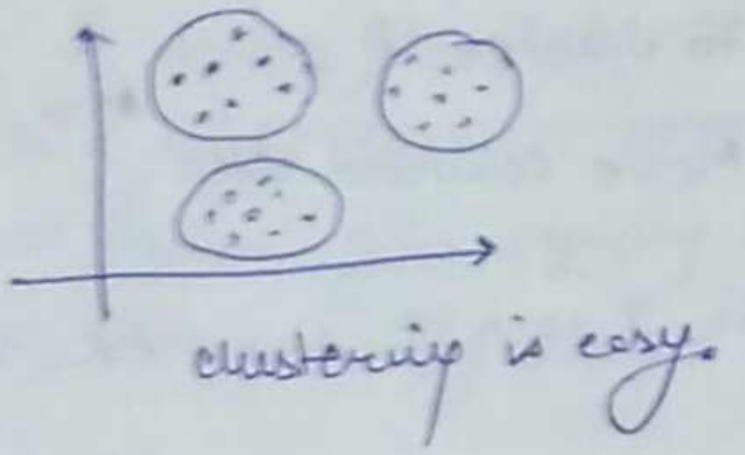
× and ⊗ are moved to new posi^n which are actually the avg of the loca^n of · and ⊙ points respectively.

Now Step 1 and 2 i.e cluster assignment and centroid movement are repeated until the centroids stop changing their position further. At this point K-means converges giving two reasonably correct clusters. Each datapoint is assigned a cluster such that it minimizes

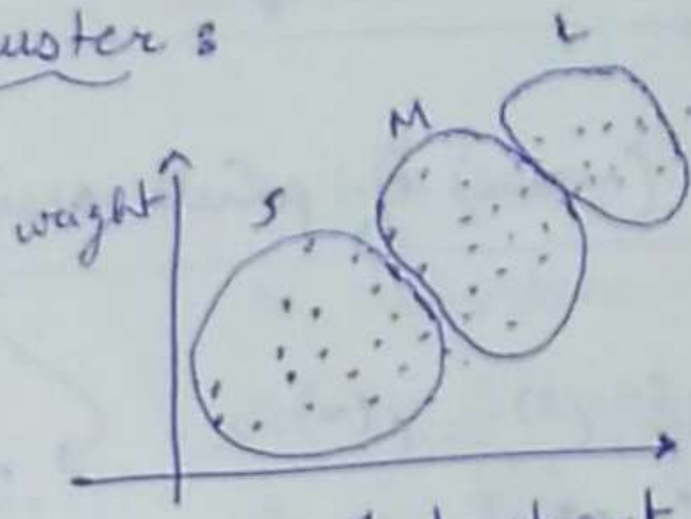
$\|x^{(i)} - \mu_k\|^2$ i.e distance b/w datapoint and cluster centroid.

If there arises a condⁿ where no datapoint is assigned to a particular cluster, then that particular cluster is eliminated or is reinitialized in case there is a need for all K clusters.

K-Means for non-separated cluster :



clustering is easy.



what about this case?

- suppose we want to make 3 clusters of Tshirt sizes 'S', 'M' and 'L'.
(e.g. of market segmentation)

K means optimization objective:

$c^{(i)}$ = index of cluster (1, 2, .. K) to which example $x^{(i)}$ is currently assigned.

μ_k = cluster centroid k ($\mu_k \in R^n$) {location}

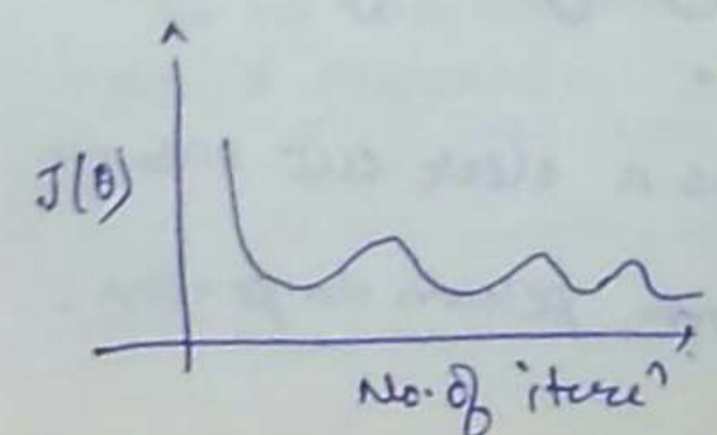
K = total no. of clusters.

$\mu_{c^{(i)}}$: cluster centroid of cluster to which example $x^{(i)}$ has been assigned.

Optimization objective : $J(c^{(1)}, \dots c^{(m)}, \mu_1, \dots, \mu_K) = \frac{1}{m} \sum_{i=1}^{m} \|x^{(i)} - \mu_{c^{(i)}}\|^2$

$\min_{c^{(1)}, \dots c^{(m)}, \mu_1, \dots \mu_K} J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K)$

This cost fⁿ is also called distortion cost fⁿ or the distortion of K Means algo.



Such a J(θ) is not possible w.r.t. iterations. There must be some bug in the code if such J(θ) is obtained.

K-means algorithm:

Randomly initialize K cluster centroids $\mu_1, \mu_2, \ldots \mu_K \in R^n$

Repeat {

Cluster assignment step

for i = 1 to m
  $c^{(i)}$ = index (from 1 to K) of cluster centroid closest to $x^{(i)}$

for k = 1 to K
  μ_k = average (mean) of points assigned to cluster k.

Move centroid step.

→ minimize J w.r.t $c^{(1)}, c^{(2)}, \ldots c^{(m)}$ holding $\mu_1, \mu_2 \ldots \mu_k$ fixed.

↳ minimize J w.r.t. $\mu_1, \ldots \mu_k$

Q. How to initialize K means and avoid local optima as well?

Ans Should have $K < m$

- Randomly pick K training examples
  Set $\mu_1, \ldots \mu_k$ equal to these K examples.

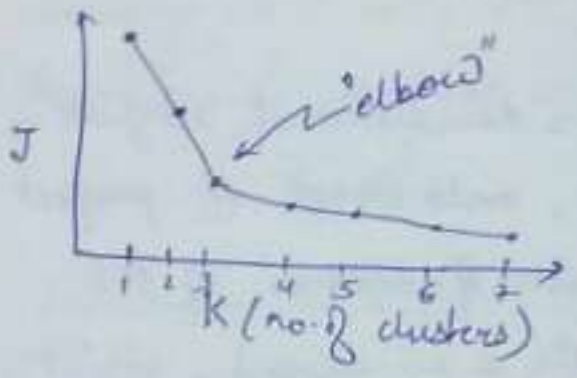
i.e $\mu_1 = x^{(1)}$ if K = 2
$\mu_2 = x^{(j)}$
i, j are some random training egs

Depending on the random initialize, K means can end up with different solutions. So, to ensure that K means end up with global and not local optima, try multiple random initializations. Then calculate the cost corresponding to each initialize and pick the one with minimum cost. This works even better when K is small (i.e 2 to 10). If K is very large, the 1st random initialize itself would lead to pretty good result and many random initialize won't make much difference.

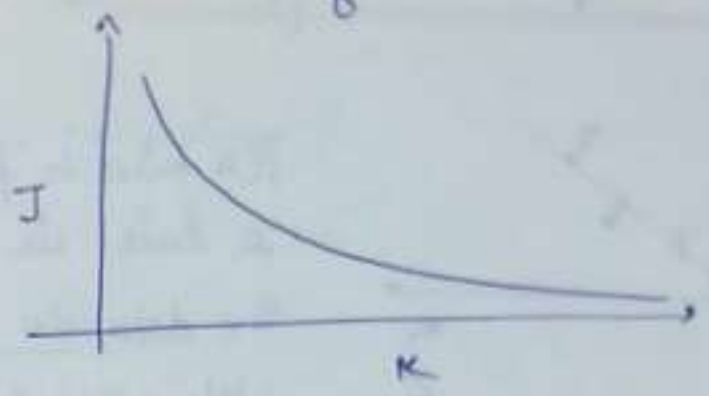
Q How to choose the no. of clusters K?

Ans The most common way is to choose it manually by looking at visualizations and K-means clustering results.
Being unsupervised learning, there isn't always a clear cut answer to the possible no. of clusters as it depends from person to person.

- Elbow method: to find a reasonable value of K.



It seems like K=3 would be a right decision to make, as distor" goes rapidly down till 3 but hardly changes after 3.



In this case, there is no clear elbow as distortion continually goes down. So, it is hard to make a choice for K in this case.

Note If J is much higher for k=5 than for k=3, that means in the run with K=5, k-means got stuck in a bad local minimum and k-means should be rerun with multiple random initializations.

- Sometimes we run K-means to get cluster to use for some later purpose. e.g. in market segmentation (T shirt e.g. taken earlier). In this case, evaluate K Means based on a metric for how well it performs for that later purpose.

* **Dimensionality Reduction**: Another unsupervised learning algorithm

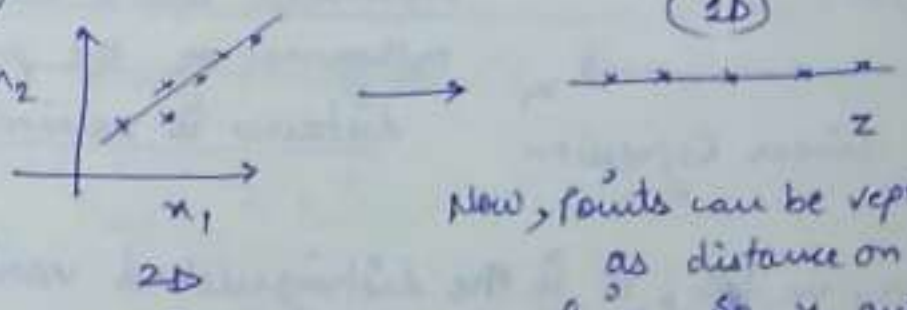
needed for

Data compression (helps speed learning algos) (helps save memory)

Visualization (reduce d data to 2D or 3D in order to visualize it.) (K=2 or 3) while visualization. → as we can maximum plot 3 features.

This dimensionality reduction is done by removing redundant or less important features or combining features to give one feature.

e.g. 2D to 1D.

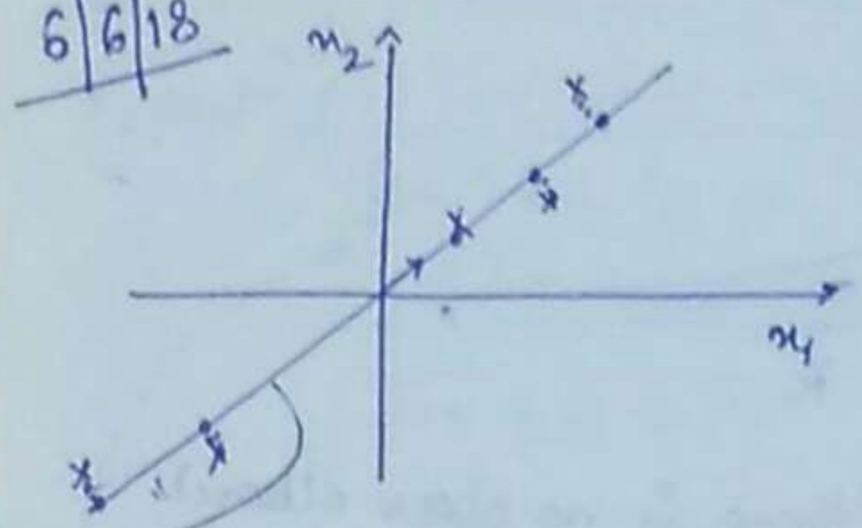


(x_1, x_2 are features)

Now, points can be represented as distance on this line. So, x_1 and x_2 are reduced to just 1 number z.

→ Principal Component Analysis (PCA) : a dimensionality reduction algorithm.

6/6/18



x : actual data points
• : projected data points
→ best line that can minimize the projection error.

PCA tries to find a lower dimensional surface → a line in this case, onto which to project the data so that the sum of squares of the little dotted line segments is minimized, i.e it tries to minimize the projection error.

Before applying, it is a standard practice to perform mean normalization & feature scaling so that features x_1 and x_2 have mean 0 and should have comparable ranges & values.

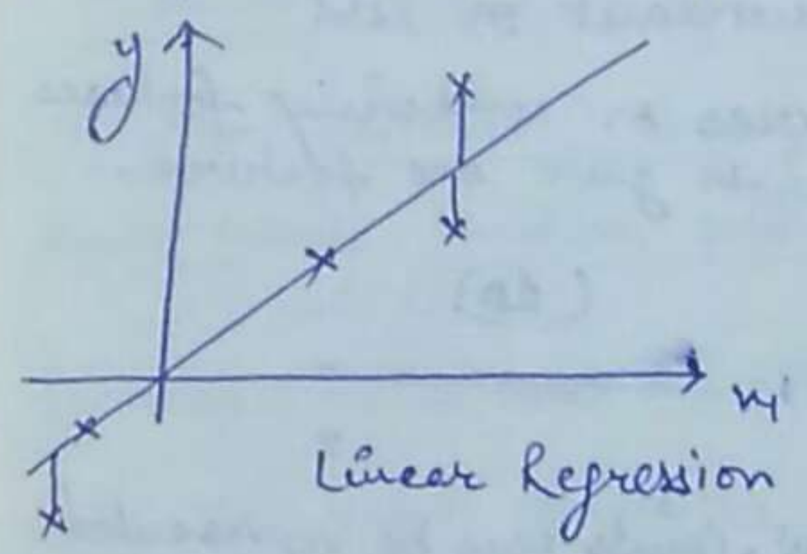
So, PCA finds a direction (a vector $u^{(1)} \in R^n$) onto which to project the data so as to minimize projection error.

Generally, we need to reduce n-dimensional to k-dimensional data where k vectors are found on which to project the data.

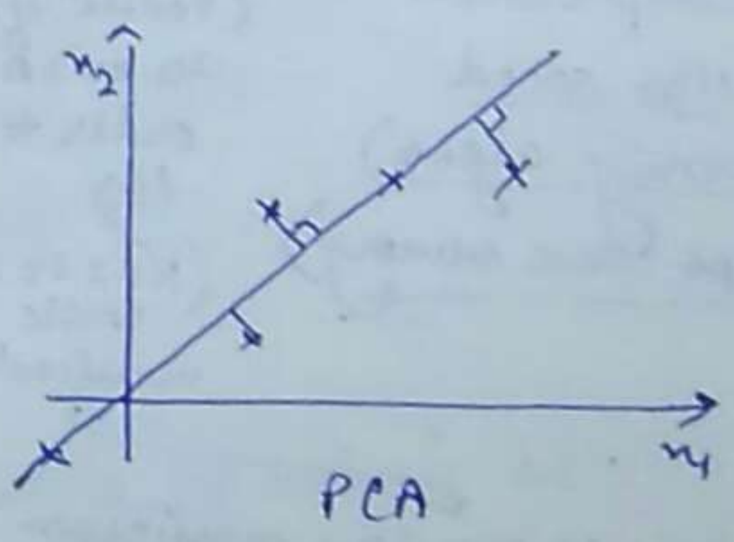
Q How PCA relates to linear regression?

Ans PCA and linear regression are 2 completely different algo though they look the same.

In linear regression, we try fit a straight line so as to minimize the square error b/w point and this straight line.



In linear regression, the vertical distance is being minimized while in PCA orthogonal or the projection distance is minimized.



Moreover, in LR, it is the distinguished variable 'y' we are trying to predict whereas in PCA there is no such y we are trying to predict rather we have a list of features $x_1, x_2 \ldots x_n$ which are all treated equally.

To apply PCA, we 1st need to do data preprocessing :

Training set : $x^{(1)}, x^{(2)}, \ldots x^{(m)}$

Preprocessing (feature scaling / mean normalize)

$\mu_j = \frac{1}{m} \sum_{i=1}^{m} x_j^{(i)}$

Replace each $x_j^{(i)}$ with $x_j - \mu_j$

If different features on different scales (eg. x_1 = size of house, x_2 = # bedrooms) scale features to have comparable range of values.

$$x_j^{(i)} = \frac{x_j^{(i)} - \mu_j}{s_j}$$

data points → $x^{(i)} \in \mathbb{R}^2 \rightarrow z^{(i)} \in \mathbb{R}^1$ ← new data points

2D to 1D reduce by PCA means :

3D to 2D . . . . : $x^{(i)} \in \mathbb{R}^3 \rightarrow z^{(i)} \in \mathbb{R}^2$ and so on.

where $z = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$ for 2D.

So, PCA needs to calculate u (i.e vectors on which to project) and z (i.e new data points in reduced dimensions)

PCA algo to reduce a n-D to k-D :

Compute covariance matrix (nxn matrix)

$$\Sigma = \frac{1}{m} \sum_{i=1}^{n} (x^{(i)})(x^{(i)})^T$$ {Σ is Greek symbol Σ to denote matrix}

Compute eigen vectors of matrix Σ -

$[U, S, V] = svd(sigma)$ {Above command for Singular value decomposition}

Note covariance matrix always satisfies a property 'Symmetric +ve semi definite' and so svd(sigma) = eig(sigma). Otherwise eig & svd are two very different functions.

$$U = \underbrace{\begin{bmatrix} | & | & & | \\ u^{(1)} & u^{(2)} & \cdots & u^{(n)} \\ | & | & & | \end{bmatrix}}_{n}$$ $U \in \mathbb{R}^{n \times n}$

and to reduce to K-dimensions, we take the first k cols i.e $u^{(1)}$ to $u^{(k)}$.

$$U_{reduce} = \begin{bmatrix} | & | & & | \\ u^{(1)} & u^{(2)} & \cdots & u^{(k)} \\ | & | & & | \end{bmatrix}$$ $U_{reduce} \in \mathbb{R}^{n \times k}$

$z = U_{reduce}^T x \in \mathbb{R}^{k \times 1}$ {Hence, we got the k vectors to project our data on}

$z^{(i)} = U^T x^{(i)}$ (for particular training eg.)

The mathematical proof that PCA gives optimized reduced dimensions is quite complex.

Q How to go back from z: (compressed representation) to x: (original representation)

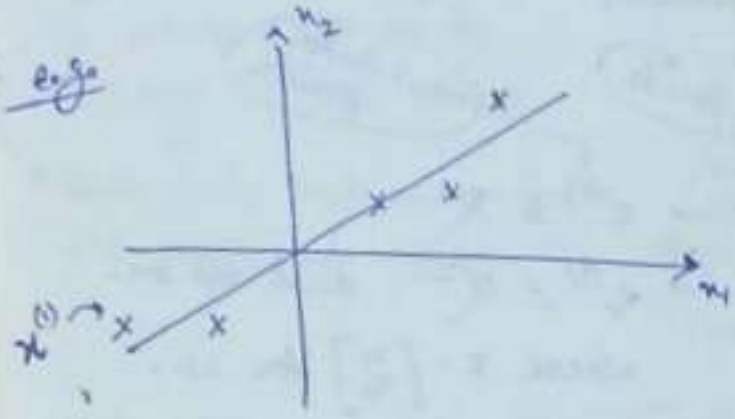
Ans We know $z = U_{reduce}^T X$

and so, $X_{approx} = \underbrace{U_{reduce}}_{n \times k} \cdot \underbrace{z}_{k \times 1}$

$n \times 1$

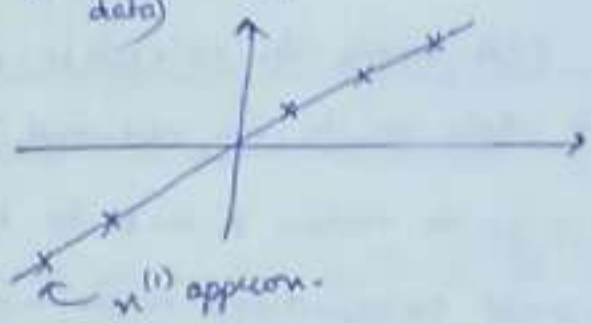
X_{approx} is closed to original X from where we started.

Eg.



PCA →

("Reconstruct" original data) ↓ back to approx X



$x^{(1)}_{approx}$

So, $x^{(1)} \approx x^{(1)}_{approx}$.

Q. How to choose K, i.e the dimension to which X need to be reduced?

Ans $K \rightarrow$ parameter of PCA algorithm.

or

no. of principal components that we've retained.

Avg. squared projection error : $\frac{1}{m} \sum_{i=1}^{m} \| x^{(i)} - x^{(i)}_{approx} \|^2$

Total variance in the data : $\frac{1}{m} \sum_{i=1}^{m} \| x^{(i)} \|^2$

(on avg. how far are our training examples from vectors, from just being all zeroes)

Typically choose k to be the smallest value so that :

① $$\frac{\frac{1}{m} \sum_{i=1}^{m} \| x^{(i)} - x^{(i)}_{approx} \|^2}{\frac{1}{m} \sum_{i=1}^{m} \| x^{(i)} \|^2} \leq 0.01$$ i.e (1%.) ★★

"99% of variance is retained". {a good way to report the K chosen}

In real life applica, many of the features are highly correlated and so its quite easy to reduce the dimensions significantly while retaining 99% variance.

S in [U, S, V] from svd (sigma) is a nxn matrix which is diagonal.

$$S = \begin{bmatrix} s_{11} & & 0 \\ & s_{22} & \\ 0 & & \ddots \\ & & & s_{nn} \end{bmatrix}$$

for given K, ① can be calculated as:

$$② - \left[1 - \frac{\sum_{i=1}^{K} s_{ii}}{\sum_{i=1}^{n} s_{ii}} \right]$$ (much simpler expression than ①)

We need to get just once the value of S to find the optimum K. We need not run PCA again & again from scratch as ② has been obtained from ①.

So, we need to check for which smallest value of K, ② ≤ 0.01

Q How PCA helps to speed up a learning algorithm?

Ans ① Supervised learning speedup

Let's say we have this data: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots (x^{(m)}, y^{(m)})$ and say $x^{(i)} \in \mathbb{R}^{10K}$ i.e we have very high dimensional data. Such huge data will surely slow down any algo be it LR or SVM or some other. Hence, we need PCA so as to speed it up withe lesser features.

Procedure:

Extract inputs: i.e unlabeled dataset = $\underbrace{x^{(1)}, x^{(2)}, \ldots, x^{(m)}}_{\downarrow PCA} \in \mathbb{R}^{10K}$

$z^{(1)}, z^{(2)}, \ldots, z^{(m)} \in \mathbb{R}^{1000}$

So,

New training set: $(z^{(1)}, y^{(1)}), (z^{(2)}, y^{(2)}) \ldots (z^{(m)}, y^{(m)})$

↳ lower dimensional feature vector than x. (obtained through PCA)

Now, this reduced dimensional data can be fed into our learning algorithm (SVM, LR, NN, etc) to have $h_\theta(z) = \frac{1}{1+e^{-\theta^T z}}$. Any new example can be mapped using PCA to give reduced feature vector z.

Note: Mapping $x^{(i)} \to z^{(i)}$ should be defined by running PCA only on the training set. This mapping can be applied as well to the e.g.s $x_{cv}^{(i)}$ and $x_{test}^{(i)}$. This is do beoz, while mapping Vreduced is calculated mean normalizn and feature scaling is done which are actually parameter learned by PCA and so this can be done on trainset only.

→ **Bad use of PCA: to prevent overfitting**

PCA might work OK but use regularizn instead.

beoz PCA does not use labels y, it only sees x and reduces it to lower dimension, hence with probability of throwing away valuable information which regularizn won't do as it takes into account the labels y unlike PCA.

* Before implementing PCA, first try running with the original/raw data $x^{(i)}$. Only if that doesn't work, implement PCA and consider using $z^{(i)}$. When you think there is a speed or memory problem, think of applying PCA, otherwise try doing with the original data only.

* WEEK 9:

Anomaly detection- (Supervised & Unsupervised learning algo)

$x^{(i)}$ = features of user i's activities

Model $p(x)$ from data

Identify unusual users by checking which have $p(x) < \epsilon$

} applin in fraud detection, manufacturing or monitoring computers in a data center, etc.

If there are too many false +ves detected, ↓ ϵ.

↓

anomalous

Gaussian Distribution (Normal distribution): to develop an anomaly detecn alg.

Say $x \in \mathbb{R}$, if x is a distributed Gaussian with mean μ, variance σ^2.

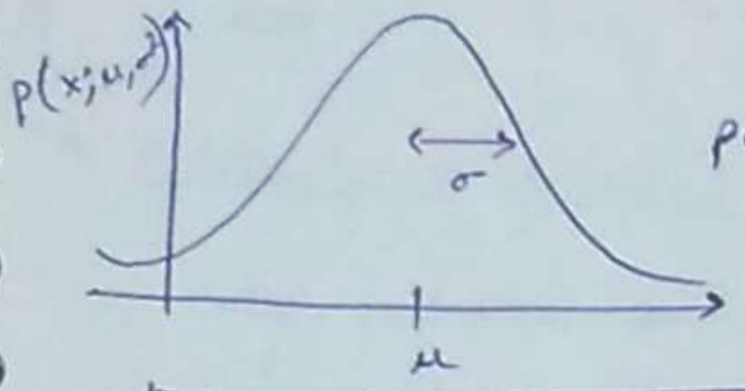
$x \sim \mathcal{N}(\mu, \sigma^2)$

↑ "distributed as" → Normal distribu^n (same as Guassian)

μ: Mean

σ^2: variance

σ: standard deviation

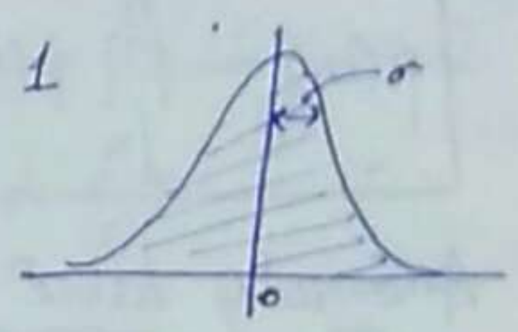


pdf of Guassian

$$\boxed{P(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)}$$

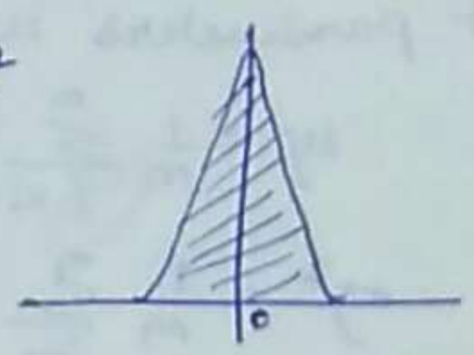
↓ Probability of x parameterized by 2 parameters μ & σ^2.

e.g. 1



$\mu = 0, \sigma = 1$

e.g. 2



$\mu = 0, \sigma = 0.5$

The area in both the examples integrate to 1 as it's the property of probability distribu^n.

- Parameter estimation:

Dataset: $\{x^{(1)}, x^{(2)}, \ldots x^{(m)}\}$ $x^{(i)} \in \mathbb{R}$

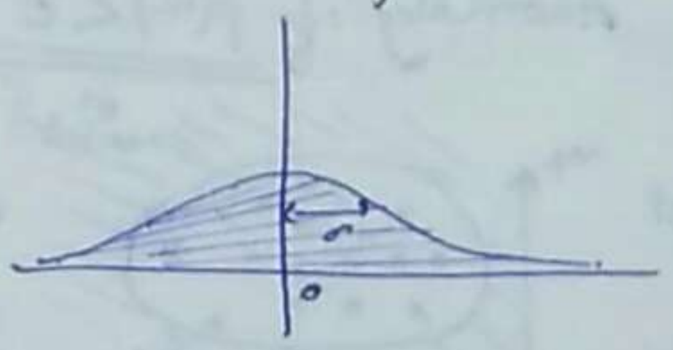
Now, we need to find μ, σ such that this dataset fits $\mathcal{N}(\mu, \sigma^2)$.

So, $\boxed{\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}}$ (means center of the dataset distribu^n)

$\boxed{\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} - \mu)^2}$

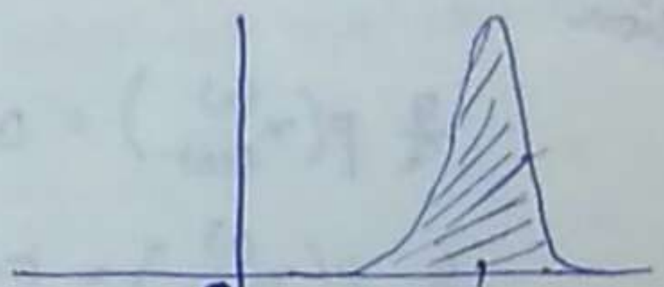
Here, μ and σ^2 are actually the <u>max likelihood estimates</u> of the parameters μ and σ^2.

e.g. 3



$\sigma = 2, \mu = 0$

e.g. 4



$\mu = 3, \sigma = 0.5$

7/6/18 <u>Density Estimation problem</u>:

Training set: $\{x^{(1)}, \ldots, x^{(m)}\}$

Each example x is $\mathbb{R}^n$

We will now model $p(x)$ from dataset to figure out imp & less imp features.

$$\boxed{p(x) = p(x_1; \mu_1, \sigma_1^2)\, p(x_2; \mu_2, \sigma_2^2)\, p(x_3; \mu_3, \sigma_3^2) \ldots p(x_n; \mu_n, \sigma_n^2)}$$

We assume that x_1 is distributed according to some Guassian distribution as $x_1 \sim \mathcal{N}(\mu, \sigma^2)$ and so on for all features.

The eq^n $p(x)$ corresponds to independence assumption on the values of the features x_1 through x_n, but the model works fine even when the features are not close to independent.

$$\boxed{p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2)}$$

- Anomaly detection algo:

1) Choose features x_i that you think might be indicative of anomalous eg.
2) Fit parameters $\mu_1, \mu_2 \cdots \mu_n, \sigma_1^2, \cdots, \sigma_n^2$

$$\mu_j = \frac{1}{m} \sum_{i=1}^{m} x_j^{(i)} \longrightarrow \mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} \quad \text{(vectorized version)}$$

$$\sigma_j^2 = \frac{1}{m} \sum_{i=1}^{m} (x_j^{(i)} - \mu_j)^2$$

3) Given new example x, compute $p(x)$;

$$p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\,\sigma_j} \exp\left(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2}\right)$$

Anomaly if $p(x) < \epsilon$.

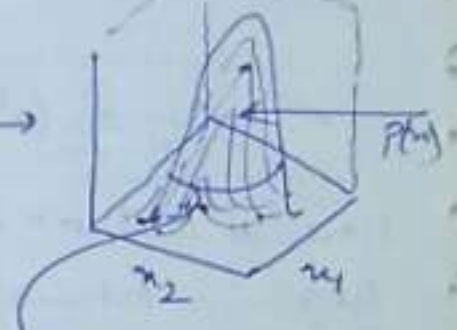
$\mu_1 = 5, \sigma_1 = 2$ (corresponding to x_1)

$\mu_2 = 3, \sigma_2 = 1$ (" " " to x_2)

Say $\epsilon = 0.02$ (chosen value)

$p(x_{test}^{(2)}) = 0.0015 < \epsilon$ Hence anomaly

$p(x_{test}^{(1)}) = 0.0426 > \epsilon$ Hence, no anomaly.



All the (x_1, x_2) pairs whose value lie below this line are anomaly.

- Evaluation of anomaly detection algo:

Real no. evaluation is important while developing a learning algorithm (choosing features, etc) for easier/simpler decision making.

eg: Aircraft engines:

10000 good engines
20 flawed "

} dataset (its OK if some bad engines slip into the 10K dataset, majority of them should be good.)

Training set : 6000 good engines. ($y=0$) → fit these to $p(x)$
CV : 2000 " " ($y=0$), 10 anomalous ($y=1$)
Test : " " ($y=0$), 10 anomalous ($y=1$)

→ Splitting of labelled and unlabelled e.g.s, with training set consisting of only features and no y, i.e training is set of unlabeled e.g.s.

Note The e.g. in CV and test set should be different.

After modelling $p(x)$ on training set,

predict $y = \begin{cases} 1 & \text{if } p(x) < \epsilon \text{ (anomaly)} \\ 0 & \text{if } p(x) > \epsilon \text{ (normal)} \end{cases}$ on CV/test e.g.

Possible evaluation metrics : F_1-score, Precision/Recall. by comparing the predicted y with actual y.

ϵ can be chosen by running on CV.

Note classification accuracy is not a good way to measure its accuracy the data is skewed and any algo predicting more no. of $y=0$ will come out as a better algo.

Q Anomaly detection v/s supervised learning ?

Ans In above e.g we saw that we use labeled data (anomalous or not), then why don't we simply use LR or NN algo. Why do we switch to anomaly detection.

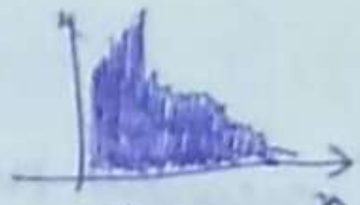
Anomaly detection	Supervised learning
→ Very small no. of +ve examples (0-20). large no. of -ve examples ($y=0$) bcoz to fit $p(x)$ we need only -ve examples (aircrafts in good condi").	→ large no. of +ve and -ve e.g.s.
→ Many diff types of anomalies. Hard for any algorithm to learn from +ve e.g.s what the anomalies look like, future anomalies may look like any of the anomalous e.g.s we've seen so far.	→ Enough +ve eg.s for algorithm to get a sense of what +ve e.g.s are like, future +ve e.g.s likely to be similar to ones in training set.

Spam email detec^ can be done by supervised learning beoz we have enough -ve examples available to capture all kind of spam emails and make the model learn which future mails would be spam and which won't be. Manufacturing defect problems are solved using anomaly detection beoz in this case we don't have much -ve examples. Similarly, fraud detc^ is anomaly detec^ problem as we don't have much data of fraud access to our system. If we do have, problem shifts to being a supervised learning problem.

Q. What features to choose to use in anomaly detec^ algorithm?

Ans We choose σ, μ considering data to be Guassian distributed.

Even if data isn't Guassian, this works fg but still it's good to do a sanitary check. If the plot doesn't look Guassian,

e.g. 

, what we do is play with different transformations of the data in order to make it more Guassian.

$\log(x)$ This would improve our results.

(Guassian)

Any transformation can be done with any of the features to transform the data to Guassian. e.g. $x_1 \leftarrow \log(x_1)$

$x_2 \leftarrow \log(x_2 + 1)$

$x_3 \leftarrow \sqrt{x_3}$, etc.

Error analysis can help find features for anomaly detec^ algorithm.

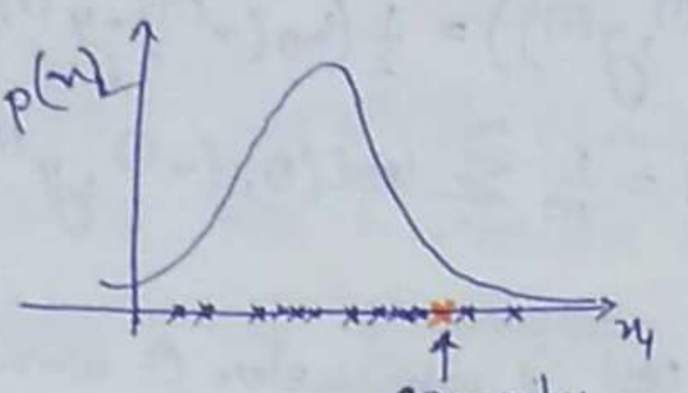
- We want $p(x)$ to be large for normal examples x.
  $p(x)$ to be small " anomalous examples x.

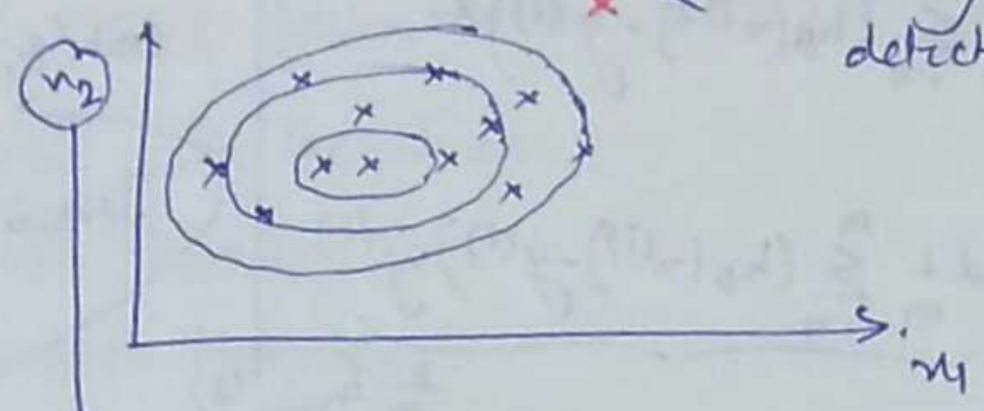
Most common problem: $p(x)$ is comparable for both normal & anomalous e.g.g, thus failing to detect anomaly.

Soln: look at the anomaly that the algo is failing to flag and see if that inspires to create some new feature. So, find something unusual about that anomaly and use that to create a new feature so that with this new feature, it becomes easier to distinguish

the anomalies from the good examples.

e.g.





Note choose features that might take on unusually large or small values in the event of an anomaly.

e.g. if x_1 = weight, x_2 = height, x_3 = BMI can be created to capture unusual health anomalies.

⇒ Multivariate Guassian Distribution : to catch anomolies that the earlier algo didn't.

8/6/18

Note Best way to get accuracy is to take a low bias learning algo and train that on a lot of data.

large datasets pose large computational problems.

Two main ideas for wiring with very big data sets < Stochastic Gradient Decent / Map Reduce

⇒ Stocastic Gradient Decent : (SGD)

Gradient descent is computationally quite expensive with large datasets so stochastic GD come into picture. This is in replacement for the general batch gradient decent (where we've looking at all of the training examples at a time.) in case of large 'm'. This SGD need not look at all training examples in one iteration rather it needs to look at only a single training example in one iteration.

⟹

Batch GD

$J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$

Repeat {

$\theta_j = \theta_j - \alpha \underbrace{\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}}_{\frac{\partial}{\partial \theta_j} J_{train}(\theta)}$

(for every $j = 0, \cdots, n$)

}

Here what is done is θ_1 is adjusted for all the training examples, then θ_2 is adjusted, then θ_3 and so on upto θ_n.

*Hence, in BGD to move towards the global minima we need to go through all the training eg.s but in SGD each training example moves us a bit closer to the global minima.

SGD

$cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2} (h_\theta(x^{(i)}) - y^{(i)})^2$

$J_{train}(\theta) = \frac{1}{m} \sum_{i=1}^{m} cost(\theta, (x^{(i)}, y^{(i)}))$

→ cost of parameter θ w.r.t the training eg $(x^{(i)}, y^{(i)})$.

Hence, this algo checks how much the hypothesis costs for training eg. $(x^{(i)}, y^{(i)})$

Steps of SGD:

i) Randomly shuffle dataset

ii) Repeat {

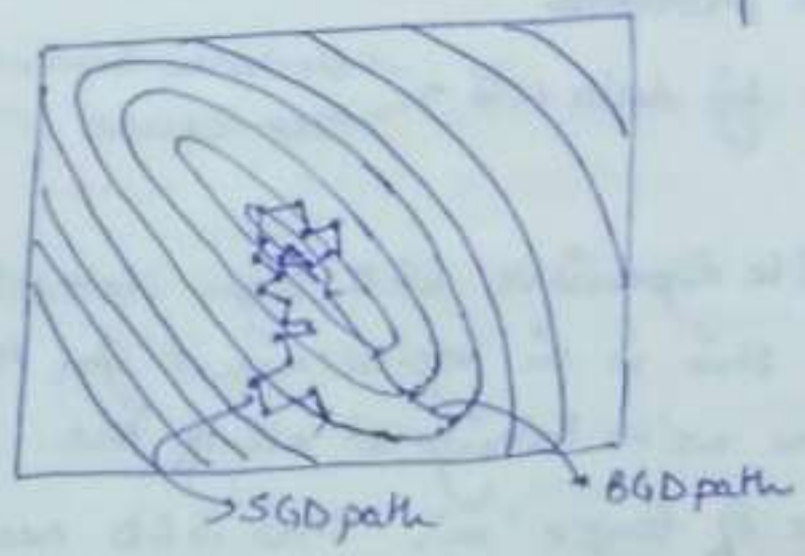
for $i = 1, \cdots, m$ {

$\theta_j = \theta_j - \alpha \underbrace{(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}}_{\frac{\partial}{\partial \theta_j} cost(\theta, (x^{(i)}, y^{(i)}))}$

} } (for $j = 0, \cdots, n$)

}

In this algo, we are basically trying to fit the parameters θ for our 1st training eg., then 2nd, then 3rd and so on adjusting θ upto the m^{th} training eg.



Note $J_{train}(\theta)$ should go down with every iteration of batch gradient descent (assuming a well tuned learning rate α) but not necessarily ∈ SGD. SGD is but much faster than BGD for large m.

⇒ Mini Batch Gradient Decent : (can sometimes work even a bit faster than SGD.)

Su BGD : use all m examples in each iteration.

SGD : use 1 example in each iteration.

mini BGD : use 'b' examples in each iteration.

↳ parameter called the mini batch size. Typical range of b is 2 to 100.

Say b = 10, m = 1000.

Repeat {

for i = 1, 11, 21, 31 . . . . 991 {

$$\theta_j = \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_\theta(x^{(k)}) - y^{(k)}) x_j^{(k)}$$

(for every $j = 0, \cdots n$)

}

}

this is faster than BGD as updates are now made after seeing 10 training egs rather than to wait for the entire training set to see an update.

Q. Why do we need to look at b examples at a time than to look at 1 training example as in SGD?

Ans Due to 'Vectorization'. Mini BGD outperforms SGD only if we have a good vectorized implementation. Sum over 10 examples can be performed in a more vectorized way allowing to partially parallelize our computation over the 10 examples, which is not quite possible in case of SGD which looks at least 1 example at a time.

One disadvantage of mini BGD is that there is now this extra parameter b which may have to be fiddled with that may take time but if a good vectorized implementation mini BGD runs even faster than SGD.

Su BGD, b = m.

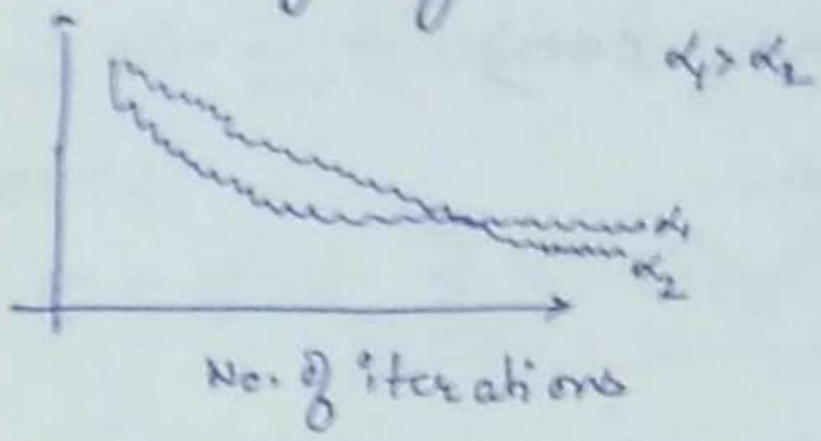
⇒ SGD convergence: to make sure that it's completely debugged and is converging okay, and how to tune α with SGD.

steps:

i) During learning, compute cost $(\theta, (x^{(i)}, y^{(i)}))$ before updating θ using $(x^{(i)}, y^{(i)})$.

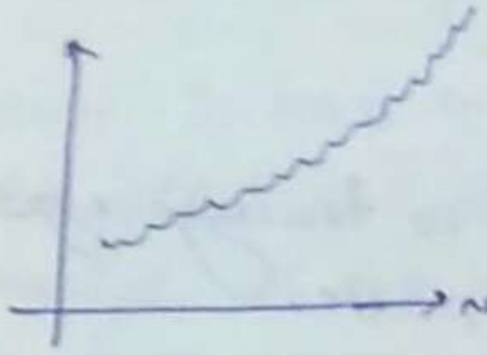
ii) Every 1000 iterations (say), plot cost $(\theta, (x^{(i)}, y^{(i)}))$ averaged over last 1000 e.g.s

processed by algorithm.

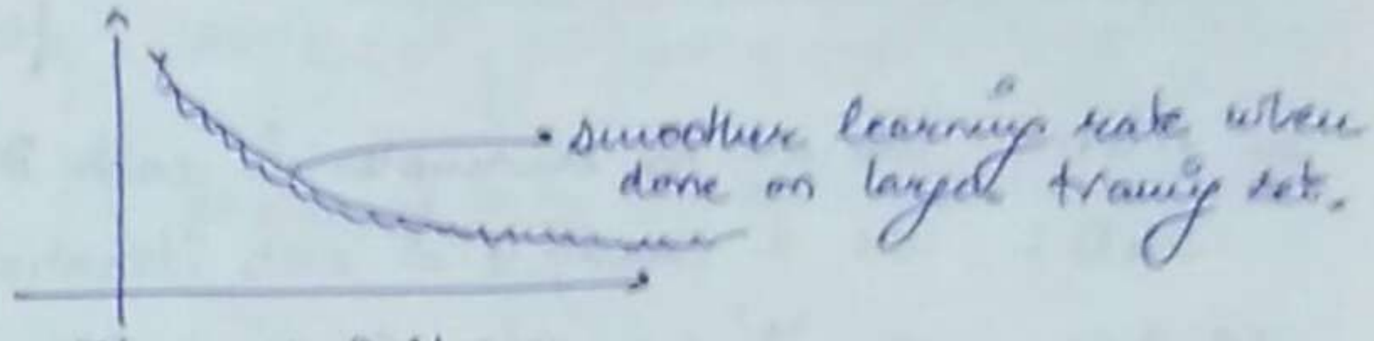


i) (little noisy bcoz they are averaged over just a 1000 egs.) They may not ↓ at every iteration.

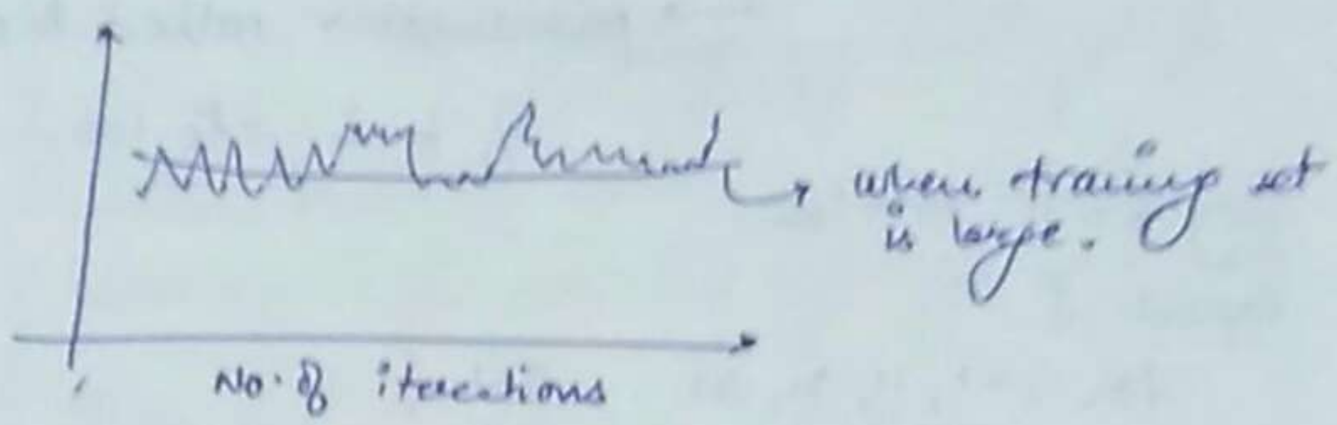
Smaller α results in smaller oscillations and so α_2 might give slightly better results.



iv) algo is diverging and smaller α needs to come.



ii)



iii) Algo isn't learning much in this case. Need either α or parameter features or algo design needs to be changed.

So, if plot seems noisy ↑ no. of training examples, if learning is not proper change α.

Note SGD doesn't really converge. It meanders around the global minimum and we get a parameter value that is hopefully close to the global but not exact at the global minimum.

Learning rate α is typically kept constant in SGD and but we can slowly ↓ α over time if we want θ to converge. [e.g. $\alpha = \frac{\text{const 1}}{\text{Iteration Number} + \text{const 2}}$.]

Doing this might lead SGD to exactly converge at the global minimum. People don't prefer doing this much as deciding upon const 1 and const 2 takes a lot of time. The above formula makes sense bcoz as algo runs, the iter no. becomes large, so α will slowly become ~~short~~ small. One can end up with a slightly better hypothesis if one ↓es α slowly to zero. Though picking very small α will slow down learning.

Note this method does not require scanning over the entire training set periodically to compute J on the entire training set, but it looks at just 1000 examples or so.

9/6/18

→ continuous learning algos which easily adapts.

→ Online learning algos: Many big companies use diff. online learning versions to learn from the flood of users. In this algo there is unlimited data available and there is no need to look at the same data more than once. This algo can adopt to changing user preferences.

eg. Product search

User searches for "Android phone 1080p camera".

Have 100 phones in store, will return 10 results.

x = features of phone, how many words in user query match name of phone, how many words in query match descrip" of phone, etc.

$y = 1$ if user clicks on link. $y = 0$ otherwise. Learn $p(y = 1 | x; \theta)$ probability that user will click on a link of a particular phone.

Predicted CLICK THROUGH RATE (CTR)

This CTR can help to show user the 10 phones that they are most likely to click on.

Other eg. choosing special offers to show user, customized sel" of news article, product recommendation. Collaborative filtering can also be used to improvise CTR.

There is no need to store the traing data and save the learned model rather our algo can continuously evolve with the a stream of data coming.

→ Map reduce and data parallelism: Some ML problems are too big to be run on one machine. So, we use map reduce approach for large scale ML. In map reduce, we split the training dataset into different subsets and send it to different machines.

Machine 1: $temp_j^{(i)} = \sum_{i=1}^{100} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$

Machine 2: $temp_j^{(i)} = \sum_{i=101}^{200} h_\theta(x^{(i)}) - y^{(i)} \cdot x_j^{(i)}$ ... and similarly for all our machines upto m examples.

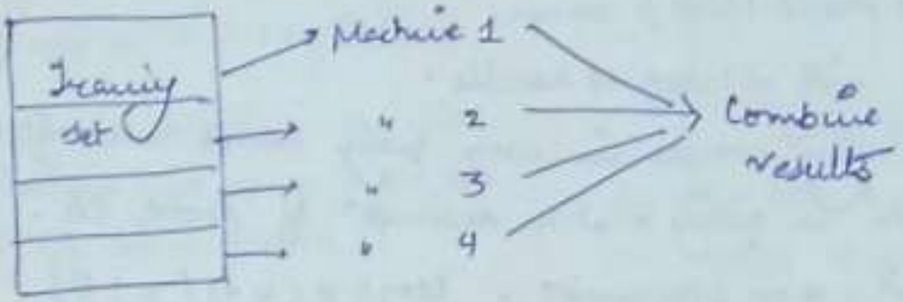
Now, each machine has only 1 quarter of the load and could do it about four times as fast.

Then all the 'temp' variables are collected and sent to a master server and update parameter $\theta_j = \theta_j - \alpha \frac{1}{m} \underbrace{(temp_j^{(1)} + temp_j^{(2)} + \dots temp_j^{(k)})}_{\downarrow}$

(for $j = 0$ to n)

$$\sum_{i=1}^{m} h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Map reduce



[Giving a 4x speed as 4 computers are 11 in working]

Note: Map reduce is a good candidate when learning algo. can be expressed as computing sums of functions over the training set.

Photo ocr pipeline:

Image → Text detection → character segmentation → character recognition

→ Artificial data synthesis:

Creating new data from scratch.

amplify the existing training set (using artificial distortions on the text dataset.)

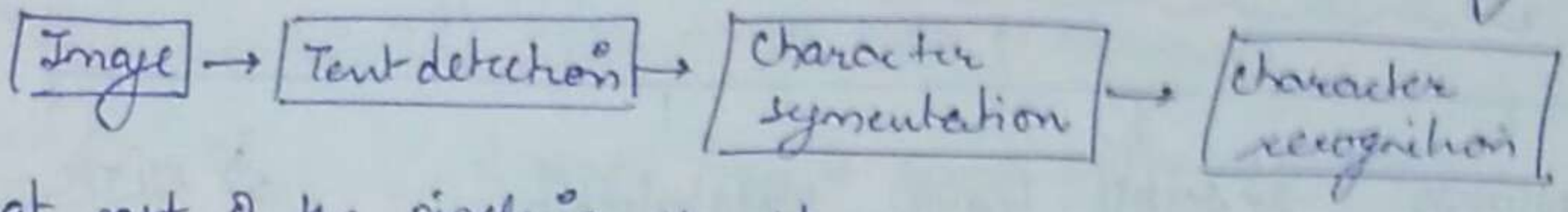
{Distortion introduced should be representation of the type of noise/distortion in the test set.} Adding meaningless noise is useless.

Note: Make sure to have a low bias classifier before expanding the training data. Keep track the no. of features in NN until we have a low bias classifier.

"Crowd source" is the method of labelling data by human labourer.

"Ceiling analysis": to give guidance on what parts of the pipeline might be best use of your time to work on.

eg:- Estimating the errors due to each component (ceiling analysis).



What part of the pipeline should you spend the most time trying to improve?

Component	Accuracy	
· Overall system	72%	↓ 17% ↑
Text detection	89%	↓ only 1% ↑
Char segmenta^	90%	↓ 10% ↑
Char recogni^	100%	

What we are doing is first we checked the accuracy for overall system. Then we checked the accuracy of the system by having 100% correct text detec^ segment (by manually segmenting the text) and found that when this segment of pipeline achieved 100% accuracy, the final results improved by 17%. Hence its would be worth spending time on improving the accuracy of this part of our segment rather than the character segmentation part bcoz even if it does with 100% perfect, the overall accuracy gain is just 1%.

'Ceiling analysis' also helps us find the max. theoritical accuracy that can be achieved if all the segments of the pipeline worked perfectly.