2020/04/14 (Machine learning) - Andrew Ng

→ supervised (right answers were given)
  - classification (discretely)
  - regression (continuous)

→ unsupervised → audio example (separating voice from noisy environment)

* Cost function: measure of accuracy of hypothesis

$$J(\theta_0,\theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2 \quad \text{(cost function)}$$

$$= \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2.$$

→ difference between the predicted value & actual value

hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$

$y = mx + c$

minimize $J(\theta_0,\theta_1)$ over θ_0,θ_1

sq. error function

* Gradient Descent: α → learning rate

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

↳ repeat this until convergence

Note: simultaneously update θ_0, θ_1

Ex: temp0 $:= \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0,\theta_1)$ (partial derivate (not derivate))

temp1 $:= \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0,\theta_1)$

θ_0 := temp0

θ_1 := temp1

} simultaneous update

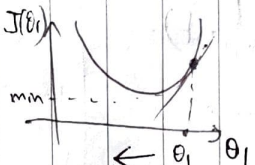
slope = tan θ

$\theta_j := \theta_j + \sqrt{\theta_0\theta_1}$ (ex: $\theta_0 = 1$, $\theta_1 = 2$)

$\theta_0 = \theta_0 + \sqrt{2} = 1 + \sqrt{2}$ ..

$\theta_1 = \theta_1 + \sqrt{2} = 2 + \sqrt{2}$ ..

Let, $J(\theta_1)$ & Here $\theta_0 = 0$ & $\theta_1 \in R$



Now,

$$\theta_j := \theta_1 - \alpha \frac{\partial}{\partial \theta_j} J(\theta_1)$$

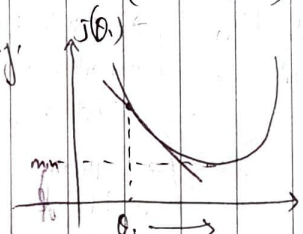
$\frac{\partial J(\theta_1)}{\partial \theta_1}$ → slope of tangent at θ_1

Positive no. / positive slope

$= \theta_1 - \alpha$ (+ve no.) , α → always +ve

$= \theta_1$ - value (decrease θ_1)

Similarly,



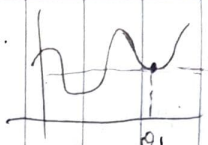
$$\theta_1 := \theta_1 - \alpha \frac{\partial J(\theta_1)}{\partial \theta_1} \quad \text{(slope -ve)}$$

$= \theta_1 - \alpha$ (-ve no.)

$= \theta_1$ + some value (increase θ_1)

α → too small: gradient descent becomes slow

→ too large: gradient descent can overshoot the minimum (fails to converge)



what happens?

$\theta_1 := \theta_1 - \alpha$ (slope at θ_1)

$= \theta_1 - \alpha \times 0$

$= \theta_1$

∴ θ_1 is not changed

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

$y = \theta_0 + \theta_1 x$

$1 = \theta_0 + \theta_1$

$\theta_0 + 2\theta_1 =$

$\theta_0 + 1\theta_1 = 0.5 \times (1 - 2\theta_1)$

$1 - 2\theta_1 + \theta_1 = 0.5$

$1 - 0.5 = 1\theta_1$

$\therefore \theta_1 = 0.5$

$\theta_0 + 2 \times 0.5 = 1$

$\theta_0 = 0$

$y = mx$

$y = 0.5$

$h_\theta(6) = \theta_0 + \theta_1 x$

$= -1 + 2x$

$= -1 + 2 \times 6$

$= 12 - 1$

$= 11$

$-2 + 0.5 \times 6$

$3 - 2$

1

$m = 0 = \frac{rise}{run} = \frac{0}{run} = 0$

(4)

* multivariable linear regression.

Hypothesis:

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

$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 \ldots + \theta_n x_n$

For convinience, $x_0 = 1$

$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$

i.e.,

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} \quad \text{and} \quad X = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \quad (n+1) \text{ dimension vector}$$

$\therefore h_\theta(x) = \theta^T X$

* Feature Scaling:

x_1 = size (0 - 5000)

x_2 = no. of bedrooms (1 - 5)

Now, x_1 and x_2 has very high difference value making contour plots difficult and making calculation of gradient descent difficult.

So, $x_1 = \frac{size}{2000}$, $x_2 = \frac{no.\ of\ bedrooms}{5}$

$0 \le x_1, x_2 \le 1$ (easy)

* Mean normalization: Replace x_i with $(x_i - \mu_i)$ except x_0

Ex:

$x_1 = \frac{size - 1000}{2000}$, $x_2 = \frac{no.\ of\ bedrooms - 2}{5}$

Now $\mu_1 = \frac{1000}{2000}$

* Multi-variable Linear Regression of cost function

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(\theta^T x - y_i)^2$$

And,

Gradient Descent,

repeat until convergence {

$$\theta_j := \theta_j - \frac{1}{m} \times \alpha \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

} for $j = 0 \ldots n$

→ features can be improved by combining multiple features. [Ex: $x_3 = x_1 \times x_2$]

* Polynomial Regression: → if hypothesis does not fit the data, we can use non-linear representations too.

→ We can change the behaviour or curve of the hypothesis function by making it quadratic, cubic, sq. root or any other form.

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

quadratic: $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2$

cubic: $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^3$

sq. root: $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 \sqrt{x_1}$

(Note: features should be considered (carefully))

* Normal Equation

→ gradient descent is one way of minimizing the cost function but this can also be achieved by normal equation.

→ In Normal Equation method, we minimize the cost function (J) by explicitly taking its derivatives with respect to θ_j (partial derivative) and setting them to 0.

→ Normal eqn allows us to find the optimum θ without iterations.

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

Normal Eqn.

→ No need to scale the features

Gradient Descent	Normal Equation
→ need to choose α	→ no need of α
→ needs many iterations	→ no need to iterate
→ works well when n is large (n → features)	→ slow if n is large
→ $O(Kn^2)$ terms (mathematical complexity)	→ $O(n^3)$ terms (mathematical complexity)

• Similar to finding minima of a curve using $\frac{dJ(\theta)}{d\theta} = 0$

* Normal Equation Non-invertibility

$(X^T X)$ → term could be non-invertible because of:

→ redundant features (when two features are very closely related) (maybe linearly dependent).

→ too many features ($m \le n$), here m is the no. of samples and n is the no. of features (in this case, use "regularization" or delete some features)

* Classification

→ To attempt classification, one method is to use linear regression and map all predictions greater than 0.5 as a 1 and less than 0.5 as 0. However this method does not work well because classification is not actually a linear function.

→ classification problem is like the regression problem, except that the values we now want to predict take on only small no. of discrete values. (Ex. 0 and 1) → binary classification problem.

→ 1 → is called positive class

→ 0 → is call negative class

(just arbitrary assumption of positive & negative)

→ $x^{(i)}$ → features of data

$y^{(i)}$ → aka label of data. $y \in \{0,1\}$

* Hypothesis Representation

→ for classification problems, there is no sense for hypothesis function to output values beyond 0 and 1. as we know that $y \in \{0,1\}$

→ To fix this, we change the hypothesis function $h_\theta(x)$ so that: $0 \le h_\theta(x) \le 1$

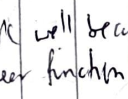
→ How to achieve this??

ans: → by plugging $\theta^T x$ into the logistic function. (sigmoid function) aka

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

let, $z = \theta^T x$

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



→ g(z) maps any real number to (0, 1).

Also in probability representation

$$h_\theta(x) = P(y=1 | x; \theta)$$

($h_\theta(x)$ gives the probability that our output is 1).

* Regression: measure of a relation betw the mean value of one variable and corresponding values of other variables (estimation & c.o.f)

• Decision Boundary

→ In order to get our discrete 0 or 1 classification we can translate the output of the hypothesis function as follows:

$h_\theta(x) \geq 0.5 \rightarrow y = 1$

$h_\theta(x) < 0.5 \rightarrow y = 0$

→ logistic function (sigmoid) characterizes the output of hypothesis function: $g(z) \geq 0.5$ when $z \geq 0$

where, $z = \theta^T x$

$= \theta_0 x_0 + \theta_1 x_1 + \cdots \theta_n x_n$

As, $z \to \infty$, $g(z) \simeq 1$

As, $z \to -\infty$, $g(z) \simeq 0$

At $z = 0$, $g(z) = 0.5$

And,

$\theta^T x \geq 0 \Rightarrow y = 1$

$\theta^T x < 0 \Rightarrow y = 0$

→ Decision boundary is the line that separates the area where $y = 0$ and $y = 1$.

Eg. $\theta^T = [5 \ -1 \ 0]$

Then $\theta^T x = 5 x_0 + (-1) x_1 + 0 \times x_2$

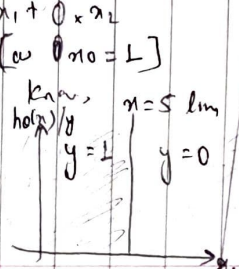
i. $\theta^T x = 5 - x_1$ [as $x_0 = 1$]

Now, for $y = 1$, we know, $\theta^T x \geq 0$

i. $5 - x_1 \geq 0$

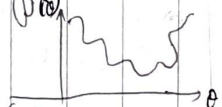
i. $-x_1 \geq -5$

i. $x_1 \leq 5$



* Cost function for logistic Regression

→ if we use the same cost function i.e $(h_\theta(x) - y^{(i)})^2$ for logistic regression, the logistic function will output a wavy curve causing many local minima (a non-convex function)



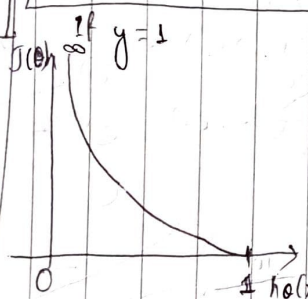
(If cost function for linear regression is used)

So, we use:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{cost}\left(h_\theta(x^{(i)}), y^{(i)}\right)$$

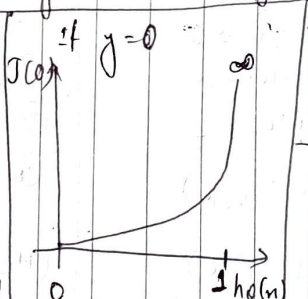
where,

$$\text{cost}(h_\theta(x^{(i)}), y^{(i)}) = \begin{cases} -\log(h_\theta(x)) & \text{for } y = 1 \\ -\log(1 - h_\theta(x)) & \text{for } y = 0 \end{cases}$$



$\text{cost}(h_\theta(x), y) \to 0$ if $h_\theta(x) = y$

$\text{cost}(h_\theta(x), y) \to \infty$ if $y = 1$ and $h_\theta(x) \to 0$



$\text{cost}(h_\theta(x), y) \to 0$ if $h_\theta(x) = y$

$\text{cost}(h_\theta(x), y) \to \infty$ if $y = 0$ and $h_\theta(x) \to 1$

→ writing the cost function in this form guarantees that $J(\theta)$ is convex for logistic regression

* Simplified cost function and Gradient Descent

→ Cost function can be compressed into a single equation such as:

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

Here,

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

& when $y = 0$, $\text{cost}(h_\theta(x), y) = -\log(1 - h_\theta(x))$

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

[entire cost function]

In vectorized implementation,

$h = g(X\theta)$ where, $g(z) = \frac{1}{e^{-z} + 1}$ = logistic function

And

$$J(\theta) = \frac{1}{m} \cdot \left[-y^T \log(h) - (1 - y)^T \log(1 - h) \right]$$

* Gradient Descent:

Repeat {

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

}

Also,

Repeat {

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

} → gradient descent for logistic regression

$\theta := \theta - \alpha/m \; X^T \left(g(X\theta) - \vec{y} \right)$ → vector representation

* Advance Optimization:

→ "Conjugate Gradient", "BFGS" and "L-BFGS" are more sophisticated and faster ways to optimize θ that can be used instead of gradient descent.

→ Use library because they are tested & highly optimized

→ For a given value θ, we need to calculate:
$J(\theta)$ and $\frac{\partial}{\partial \theta_j} J(\theta)$

→ In octave, we can write a function to return these values as:

```
function [jval, gradient] = costFunction(theta)
    jval = % code to compute J(theta)
    gradient = % code to compute derivative of J(theta)
end
```

→ Octave has "fminunc()" optimization algorithm along with "optimset()" function that creates an object containing the options we want to send to "fminunc()"

Code:
```
options = optimset('GradObj', 'on', 'MaxIter', 100);
initialTheta = zeros(2,1);
[optTheta, functionVal, exitFlag] = fminunc(...
    @costFunction, initialTheta, options);
```

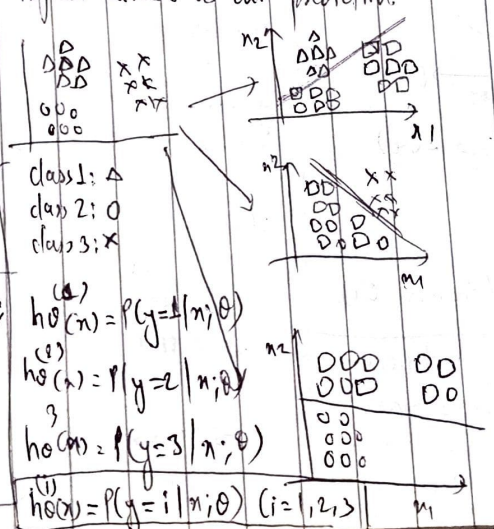
* Multiclass classification
(One vs. all) (One vs. rest)

→ For multi-class, $y \in \{0, 1, \ldots n\}$

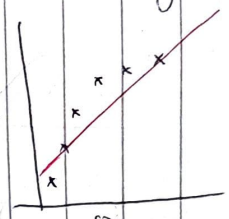
→ To achieve this, we divide our problem into $(n+1)$ binary classification problem (+1 because index starts at 0)

→ We are basically choosing one class and lumping all the others into a single second class.

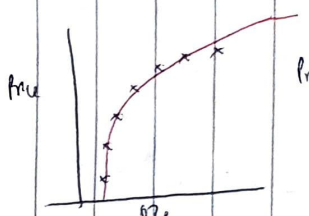
→ We do this repeatedly, applying binary logistic regression to each case and then use the hypothesis that returned the highest value as our prediction.



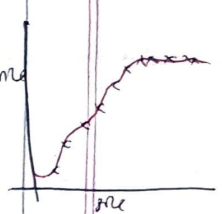
* Over-fitting



$h_\theta(x) = \theta_0 + \theta_1 x_1$
Under-fitting (High-bias)



$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x^2$
"Just right"



$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$
(Overfitting)
(High-variance)

→ Overfitting: If we have too many features, the learned hypothesis may fit the training set very well ($J(\theta) = \frac{1}{2m}\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \approx 0$), but fail to generalize to new examples.

→ Overfitting is caused by a complicated function that creates a lot of unnecessary curves and angles unrelated to the data.

→ Underfitting: when hypothesis function maps poorly to the trend of data, usually caused by function that is too simple or uses too few features.

→ Address Overfitting: (i) Reduce the no. of features:
→ manually select the features to keep
→ use a model selection algorithm

(ii) Regularization:
→ keep all features but reduce the magnitude of parameters θ_j
→ regularization works well when we have lot of slightly useful features

* Regularization:

→ If we have overfitting from our hypothesis function we can reduce the weight that some of the terms in our function carry by increasing their cost

→ Let, $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

to make $h_\theta(x)$ quadratic, we have to eliminate the influence of θ_3 & θ_4.

$$\min_\theta \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + 1000 \cdot \theta_3^2 + 1000 \cdot \theta_4^2$$

→ The last two terms increases the cost of θ_3 and θ_4 significantly and now to decrease the overall cost of the hypothesis, θ_3 and θ_4 should be greatly reduced. This results in a simple quadratic function where, $\theta_3 \approx 0$ and $\theta_4 \approx 0$

modified cost function **

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

where, λ is the regularization parameter

$\lambda \sum_{j=1}^{n} \theta_j^2$ is regularization component/term

modified gradient descent

Repeat {

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

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

$j \in \{1, 2 \cdots n\}$ (add θ_0 update separately with 1)

}

→ term $\left(1 - \alpha \frac{\lambda}{m}\right)$ is always less than 1

→ term $\left(\frac{\lambda}{m} \theta_j\right)$ performs the reg. regularization

modified normal equation

$$\theta = \left(X^T X + \lambda \cdot L\right)^{-1} X^T y$$

where, $L = \begin{bmatrix} 0 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix}_{(n+1) \times (n+1)}$

(normal eq^n is alternative to gradient descent's iterative way)**

(Note) → If $m < n$, then $X^T X$ is non-invertible (having no inverse). However, if we add the term $(\lambda \cdot L)$, then $(X^T X + \lambda \cdot L)$ becomes invertible

* Regularization of logistic Regression

→ logistic regression can be regularized similarly as the linear regression

Cost function

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

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

Gradient Descent

Repeat {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_0^{(i)}$$

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

}

* Neural Networks

→ At a very simple level, neurons are basically computational units that take inputs (through dendrites) as electrical inputs (called "spikes") that are channeled to outputs (axon).

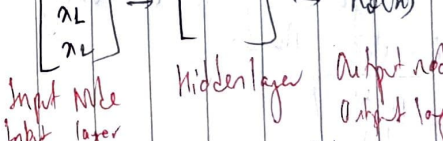
→ In our model, our dendrites are like the input features $x_1, x_2 \cdots x_n$ and the output is the result of our hypothesis function.

→ In neural networks, x_0 input node is sometimes called the biased unit ($x_0 = 1$, always)

→ In neural networks, we use the same logistic function as in classification $\left(\frac{1}{1 + e^{-\theta^T x}}\right)$, sometimes, we call it a sigmoid (logistic) activation function.

("theta" are sometimes called "weights".

A simple representation of neural networks:



→ Intermediate/hidden layer nodes $a_0^{(2)} \cdots a_n^{(2)}$ are called activation units.

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

$\Theta^{(j)}$ = matrix of weights controlling function mapping from layer j to layer $j+1$

Suppose, a one hidden layer neural network represented as:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \rightarrow \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix} \rightarrow h_\theta(x)$$

$a_1^{(2)}, a_2^{(2)}, a_3^{(2)} \rightarrow$ activation units for layer 2.

Here, (bias unit)

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

$$\vdots$$

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

&, (bias unit)

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

Here, $\theta^{(1)} = (3 \times 4)$ matrix

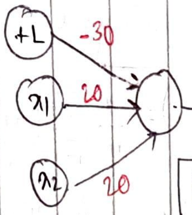
$\theta^{(2)} = (1 \times 4)$ matrix

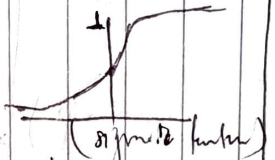
* Each layer gets its own matrix of weights, $\theta^{(j)}$

* Note: If a network has s_j units in layer j and s_{j+1} units in layer $(j+1)$ then $\theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$

* (+1 term comes from the addition of bias unit)

logical AND of x_1, x_2.





(sigmoid function)

$$h_\theta(x) = -30x_0 + 20x_1 + 20x_2$$

$\theta^{(1)} = [-30 \;\; 20 \;\; 20]$ (AND)

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	$g(10) \approx 1$

$\theta^1 = [10 \; -20 \; -20]$ (NOR)

$\theta^1 = [-10 \;\; 20 \;\; 20]$ (OR)

* We can combine these to get the XNOR logical operator (which gives 1 if x_1 and x_2 are both 0 or both 1)

→ for the transition between the first & the second layers, we'll use $\theta^{(1)}$ matrix that combines the values for AND & NOR:

$$\theta^1 = \begin{bmatrix} -30 & 20 & 20 \\ 10 & -20 & -20 \end{bmatrix}$$

& for 2nd the 3rd layers we use,

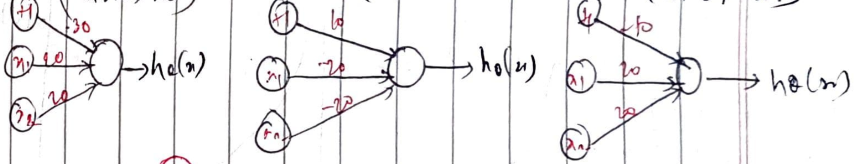
$$\theta^2 = [-10 \;\; 20 \;\; 20]$$

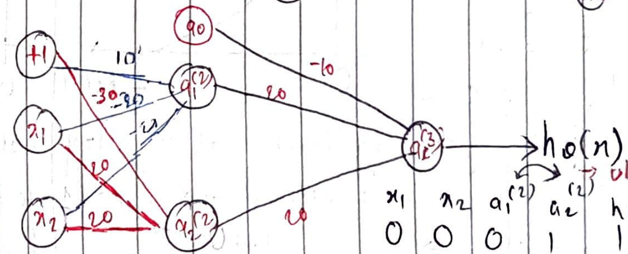
Here,

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

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

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





→ ulto lekhe leo aur mark

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

* Cost function for Neural Networks

Here,
L = total no. of layers in the network
s_L = no. of units (except bias unit) in layer L
K = no. of output units/classes
$h_\Theta(x)_K$ = hypothesis that results in the K^{th} output

→ In Neural Networks, we may have multiple outputs. (multiple $h_\Theta(x)$)
→ We know, for logistic regression (regularized), the cost function is as:

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

For Neural Networks:

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

→ a few nested summations have been added to account for multiple output nodes
→ In the first part of the equation, before the square brackets, we have an additional nested summation that loops through the number of output nodes.
→ In the regularization part, after the sq. brackets, we must account for multiple theta matrices. The number of columns in our current theta matrix is equal to the number of nodes in our current layer (including the bias unit).
→ The number of rows in our current theta matrix is equal to the number of nodes in the next layer (excluding the bias unit).

Note:
→ The double sum simply adds up the logistic regression costs calculated for each cell in the output layer.
→ The triple sum simply adds up the squares of all the individual Θs in the entire network.
→ The i in the triple sum does not refer to training example i.

* Back-propagation Algorithm

→ Back propagation is a neural network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression. Our goal is to compute: $\min_\Theta J(\Theta)$
→ Concretely, we want to minimize our cost function $J(\Theta)$ using an optimal set of parameters in theta.

$$\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta)$$

• Steps: We are given: Training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$
→ set $\Delta_{ij}^{(l)} = 0$ for all (l, i, j) (Δ_{ij} becomes matrix full of zeros)

For • training example $t = 1$ to m:
① Set $a^{(1)} := x^{(t)}$
② Perform feed-forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$
③ Using $y^{(t)}$, compute $\delta^{(L)} = a^{(L)} - y^{(t)}$
Here, L = total no. of layers
$a^{(L)}$ = vector of outputs of the activation units for the last layer
$y^{(t)}$ = correct outputs in y
→ an error value is simply the difference between our actual results from the last layer and the correct outputs in y.

4. Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$

using $\delta^{(l)} = \left((\Theta^{(l)})^T \delta^{(l+1)} .* a^{(l)} .* (1 - a^{(l)})\right)$

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

5. $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

In vectorized form,

$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

6. $D_{ij}^{(l)} := \frac{1}{m} \left(\Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} \right)$, if $(j \neq 0)$

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

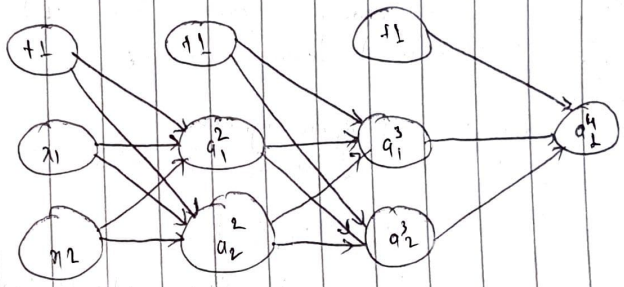
Here, capital-delta matrix D is used as an "accumulator" to add up values as we go along and eventually compute our partial derivative. Thus:

$$\frac{\partial J(\Theta)}{\partial \Theta_{ij}^{(l)}} = D_{ij}^{(l)}$$

* → Intuitively, $\delta_j^{(l)}$ is the "error" for $a_j^{(l)}$ (unit j in layer l). More formally, the delta values are actually the derivative of the cost function.

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(t), \quad \text{Recall, } \text{cost}(t) = y^{(t)} \log(h_\theta(x^{(t)})) + (1 - y^{(t)}) \log(1 - h_\theta(x^{(t)}))$$

→ Recall that the derivate is the slope of a line tangent to the cost function, so the steeper the slope the more incorrect we are.



Note:

$\delta_j^{(l)}$ = "error" of cost for $a_j^{(l)}$ (unit j in layer l)

→ Back-propagation is similar to forward propagation but from right to left

Here,

$\delta_1^{(4)} = a_1^{(4)} - y$

Now,

$\delta_1^3 = \Theta_{11}^3 \cdot \delta_1^{(4)}$ and $\delta_2^3 = \Theta_{12}^3 \delta_1^{(4)}$

$\delta_1^2 = \Theta_{11}^2 \delta_1^3 + \Theta_{21}^2 \delta_2^3$ and $\delta_2^2 = \Theta_{12}^2 \delta_1^3 + \Theta_{22}^2 \delta_2^3$ ...

* Gradient Checking.

→ Gradient Checking ensures that our back propagation works as intended.

→ We can approximate the derivative of our cost function with:

$$\frac{\partial}{\partial\theta}(J(\theta)) \approx \frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon}$$

Here ε → Epsilon, a small real number
$\approx (10^{-4})$ - guarantees that our math works properly

If ε is too small, we can end up with numerical problems.

→ With multiple theta matrices we can write:

$$\frac{\partial}{\partial\theta_j}(J(\theta)) \approx \frac{J(\theta_1, \dots, \theta_j+\varepsilon, \dots, \theta_n) - J(\theta_1, \dots, \theta_j-\varepsilon, \dots \theta_n)}{2\varepsilon}$$

→ Run the gradient checking only till you have verified the back propagation, since the code to compute gradApprox. is very slow.

* Zero Initialization:

→ Initializing all parameters (thetas) as 0 at first.

→ This causes an error called symmetry as thetas for a single activation unit links to previous units/nodes/input nodes, will end up taking the same values and this causes redundancy and not a good representation of a problem/model. (also all nodes will take upon the same value repeatedly)

* Random Initialization (symmetry breaking)

→ Initialize each $\theta_{ij}^{(l)}$ to a random value in $[-\varepsilon, \varepsilon]$
ie $[-\varepsilon \le \theta_{ij}^{(l)} \le \varepsilon]$

Ex: `theta1 = random(10,11) * (2* INIT_EPSILON) - INIT_EPSILON`

* Putting it Together * (Conclusion)

→ First, pick a network architecture, choose the layout of your neural network including how many hidden units in each layer and how many layers in total you want to have.

- No. of input units = dimension of features $x^{(i)}$
- No. of output units = no. of classes
- No. of hidden units per layer = usually more the better (must balance the cost of computation as it increases with more hidden units)
- Default: 1 hidden layer. (if you have more than 1 hidden layer, then it is recommended that you have the same no. of units in every hidden layer).

→ Training a Neural Network:

- randomly initialize the weights
- implement forward propagation to get $h_\theta(x^{(i)})$ for any $x^{(i)}$
- implement the cost function
- implement the back propagation to compute partial derivatives
- use gradient checking to confirm that your back propagation works. Then disable gradient checking.
- use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.

> For i = m:
> perform forward propagation & back propagation using example $(x(i), y(i))$
> (get activations $a^{(l)}$ and delta terms $\delta(l)$ for $l = 2, \dots L$

↳ When we perform forward & backward propagation, loop on every training example.

Note: ideally we want $h_\theta(x^{(i)}) \approx y^{(i)}$. This will minimize our cost function. However, we should keep in mind that $J(\theta)$ is not convex and thus we can end up in a local minimum instead.

* Evaluating the hypothesis:

→ What should we try next to solve the unacceptably large errors?!

- get more training examples - fixes high variance
- try small sets of features - fixes high variance
- try getting additional features - fixes high bias
- try adding polynomial features - fixes high bias
- try decreasing λ fixes high bias
- try increasing λ fixes high variance

→ Diagnostic ?! → a test that you can run to gain insight what is/isn't working with a learning algorithm, and gain guidance as to how best to improve its performance.

→ Diagnostic takes time to implement, but doing so can be a great use of your time.

→ once we have trouble-shot the errors in our hypothesis, we now evaluate it. A hypothesis may be over-fitted and thus to evaluate a hypothesis, given a dataset of training examples, we can split the data into two sets: a training set and a test set. (Typically, training set : 70%, test set : 30%.

* Model Selection & Train/validation/Test sets

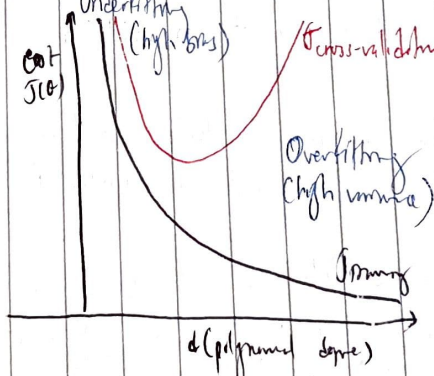
→ The error of your hypothesis as measured on the data set with which you trained the parameters will be lower than the error on any other data set.

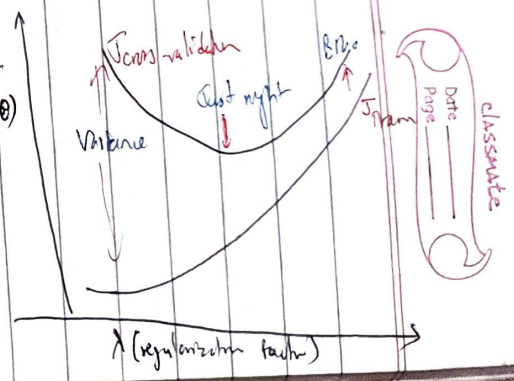
→ Given many models with different polynomial degrees, we can use a systematic approach to identify the best function. In order to choose the model of your hypothesis, you can test each degree of polynomial and look at the error result.

→ One way to break down is into the three sets:
- Training set 60%
- Cross validation set 20%
- Test set 20%

→ Now → optimize parameters in Θ using the training set
→ find the polynomial degree d with the least error using the cross validation set
→ estimate the generalization error using the test set with $J_{test}(\Theta^{(d)})$, where (d = theta from polynomial with least error)

* Bias Vs. Variance

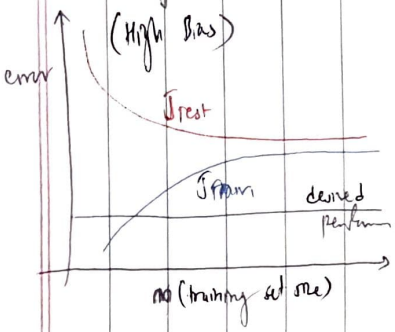


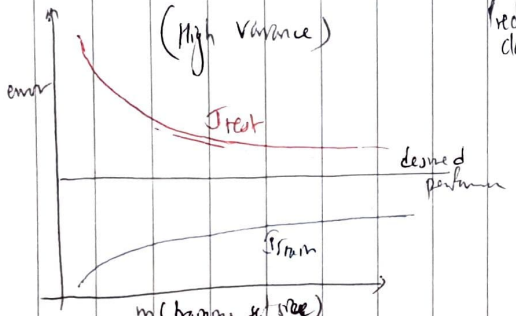


* Learning Curves

→ Training an algorithm on a very few no. of data points (2, 3 etc) will easily have 0 errors because we can always find a quadratic curve that touches exactly these number of points. Hence:

- as training sets get larger, the error for ~~the~~ a quadratic function increases.
- the error value will plateau out after a certain m, or training set size.





* Error Analysis

→ Start with a simple algorithm that you can implement quickly. Implement it and test it on your cross-validation data.

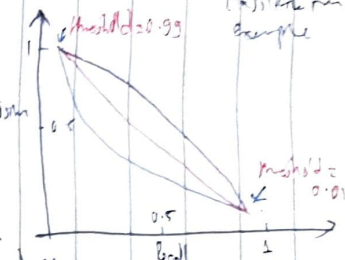
→ Plot learning curves to decide if more data, more features, etc are likely to help.

→ Error analysis: manually examine the examples (in cross validation set) that your algorithm made errors on. See if you spot any systematic trend in what type of examples it is making errors on.

* Skewed Classes: have very large no. of one type of data than other. Ex: Cancer classification example

```
function y = predictCancer(x)
    y = 0; %ignore x!
return
```

→ outputs 1 no matter what.



		Actual Class: 1	Actual Class: 0
Predicted class	1	True positive	False positive
	0	False negative	True negative

* Precision: $\frac{\text{True positives}}{\text{\# predicted positives}} = \frac{\text{True positives}}{\text{True positives + false positives}}$

* Recall: $\frac{\text{True positives}}{\text{\# actual positives}} = \frac{\text{True positives}}{\text{True positives + false negatives}}$

* Trading off Precision & Recall:

→ Predict 1 if $h_\theta(x) \geq$ threshold

Logistic regression: $0 \leq h_\theta(x) \leq 1$

Ex: → Suppose we want to predict $y = 1$ (cancer) only if very confident. (High precision, low recall) $[h_\theta(x) \geq 0.9]$

→ Suppose we want to avoid missing too many cases of cancer (avoid false negatives). (High recall, low precision) $[h_\theta(x) \geq 0.3]$

→ Measuring and deciding on values regarding both Precision & recall is non-intuitive and difficult. So, F_1 score is used.

→ F_1 Score (combines the precision/recall nos.)

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