Machine Learning

Nagubandi Krishna Sai
MS20BTECH11014

June 2021

Contents

Chapter 1

Fundamentals of Machine Learning

1.1 Supervised Learning

Supervised Learning gives "correct answers", the output values are same as real life values.
In Supervised Learning, we are given a set of data and we know what our correct output should look like, having an idea that there is relationship between the input and the output.
Supervised Learning problems has two types of problems,

1. Regression.
2. Classification.

1.1.1 Linear Regression

In regression type of problems, we are trying to predict results within a continuous output, means that we are trying to map input variables to some continuous function.
Linear regression has real-valued output, but the output will be same or near valued to the actual output.

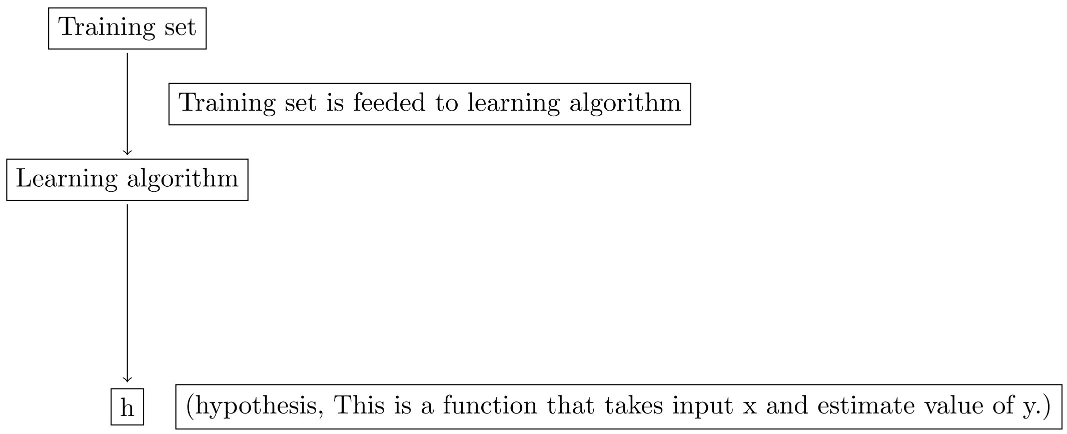
1. m = Number of training examples.
2. x's = "input" variable (or) feature.
3. y's = "output (or) target" variable.
4. (x,y) = one training example.
5. ($x^{(i)}$,$y^{(i)}$) = i^{th} training example.

Size of feet^2(x)	Price(\$) in 1000's (y)
2104	460
1416	232
1534	315
852	178
$\vdots$	$\vdots$

Table 1.1: Training set of housing prices.

Example :

1. m = 47.
2. $x^{(1)} = 2104$ and $y^{(1)} = 460$.
3. $x^{(2)} = 1416$ and $y^{(2)} = 232$.


Training set
Training set is feeded to learning algorithm
Learning algorithm
h
(hypothesis, This is a function that takes input x and estimate value of y.)

1.1.2 Hypothesis for Linear Regression

$$h_\theta(x) = \theta_0 + \theta_1.x \tag{1.1}$$

$$\theta_i\text{'s} = \text{parameters.}$$

This type of hypothesis mode is called "Linear regression with one variable" (or) "Univariate linear regression."

1.1.3 Cost function for Linear Regression

Cost function helps us know that how well to fit the best possible straight line over the given data.

Q$\rangle$ *How to choose* θ_i's ?

1. Choose θ_i 's so that $h_\theta(x)$ is close to y for our training example (x,y).
2. minimise θ_0,θ_1 so, that [$h_\theta(x)$ - y] is small.

$$J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}[h_\theta(x^{(i)}) - y^{(i)}]^2 \tag{1.2}$$

J(θ_0,θ_1) = Cost function (or) Squared error function.

1.1.4 Gradient descent for Linear Regression

Gradient descent is used to minimise cost function(J) in linear regression.
Gradient descent is used in many areas to minimise many functions in ML/AI.
Gradient descent algorithm,

$$\text{Repeat until convergence (minimum)}\left\{\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta_0, \theta_1),\ for\ j = 0, j = 1.\right. \tag{1.3}$$

1. := is Assignment operator.
2. α is learning rate.
3. $\frac{\partial}{\partial\theta_j}J(\theta_0, \theta_1)$ is derivative.

Gradient descent is nothing but the derivative of the Cost function.

$$Slope\ of\ cost\ function\ curve = \frac{\partial J(\theta_1)}{\partial\theta_1},\ when\ \theta_0 = 0. \tag{1.4}$$

Learning rate,

1. If α is too small, gradient descent can be slow. After many such operations(can be infinite times), the 'θ_1' could reach "global minimum".
2. If α is too large, gradient descent can overshoot the minimum. It may "fail to converge (or) even diverge".
3. If θ_1 is at the local optima itself when we started or taken θ_1, then there is no use of "α (or) gradient descent".

1.1.5 Linear Regression for multivariables

Hypothesis,

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n. \tag{1.5}$$

In the total context of Supervised learning, hypothesis is just predicting the output.
For convenience of notation, declare x$_0$ = 1 ($x_0^{(i)} = 1$).

$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \epsilon\ \Re^{n+1} \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} \epsilon\ \Re^{n+1}$

The above matrix is 0 - indexed.

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n . h_\theta(x) = \theta^\top x. \tag{1.6}$$

Cost function,

$$J(\theta) = J(\theta_0, \theta_1, \theta_2, ..., \theta_n) = \frac{1}{2m}\sum_{i=1}^{m}[h_\theta(x^{(i)}) - y^{(i)}]^2 \tag{1.7}$$

Gradient descent,

$$\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta_0, \theta_1, \theta_2, ..., \theta_n) \tag{1.8}$$

$$\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta) \tag{1.9}$$

$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}[h_\theta(x^{(i)}) - y^{(i)}]x_j^{(i)} \tag{1.10}$$

Feature scaling,
Get every feature into approximately -1$\leq x_i \leq 1\ range.$
Mean normalization,
$Replace x_i\ with\ x_i - \mu_i\ to\ make\ features\ have\ approximately\ zero\ mean.$

1.1.6 Polynomial regression

Hypothesis,

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3. \tag{1.11}$$

Housing price prediction
$x_1 = size$
$x_2 = (size)^2$
$x_3 = (size)^3$

Table 1.2: Features be-like in Polynomial regression.

1.1.7 Normal Equation

Intuition,

$$\tfrac{d}{d\theta}J(\theta) = 0.$$

Cost function,

$$\theta \in \Re^{n+1},\ J(\theta_0, \theta_1, \theta_2, ..., \theta_n) = \frac{1}{2m}\sum_{i=1}^{m}[h_\theta(x^{(i)}) - y(i)]^2 \qquad (1.12)$$

$$\frac{\partial}{\partial\theta_j}J(\theta) = 0,\ (solve\ for\ \theta_0\ , \theta_1\ , ...\ , \theta_n) \qquad (1.13)$$

Example,

	Size (feet^2)	No.of Bed rooms	No.of floors	Age of home (years)	Price($1000)
x_0	x_1	x_2	x_3	x_4	y
1	2104	5	1	45	460
1	1416	3	2	40	232
1	1534	3	2	30	315
1	1852	2	1	36	178

Table 1.3: Sample Training set for Multi-Variate Linear regression.

1. n = number of Features.
2. $x_j^{(i)}$ = value of j in the i^{th} training example.
3. $x_{(i)}$ = the input(features) of the i^{th} training example.

$$X = \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{bmatrix} \qquad Y = \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}$$

X is m×(n+1)-dimensional matrix and Y is a m-dimensional vector.

$$\theta = (X^\top X)^{-1} X^\top Y \tag{1.14}$$

The above θ value is optimal θ value.
For Normal equation method, then no need to use **feature scaling.**
We use 'Gradient descent' and 'Normal equation' methods to minimise cost function.

Gradient descent	Normal equation
1⟩ Need to choose 'α'.	1⟩ No need to choose 'α'.
2⟩ Need many iterations.	2⟩ Don't need to iterate.
3⟩ Works well even, when 'n' is large. (n¿10000)	3⟩ Need to compute n$\times$ n matrix inverse (X⊤X)$^{-1}$
	4⟩ Works Now if n is very large.

Table 1.4: Why should we use the particular method? Advantages and Disadvantages of two methods.

$$\theta = (X^\top X)^{-1} X^\top Y \tag{1.15}$$

Q⟩ What is $X^\top X$ is non-invertible(singular/degenerate) ?
Reasons,

1. Redundant features (linearly dependent)
   - x_1 = Size in $feet^2$
   - x_2 = Size in m^2
   - $x_2 = (3.28)^2\ x_1$, 1m = 3.28feet.
2. Too many features. (m$\leq$n)
   - m = 10
   - n = 100, $\theta \ \epsilon\ \Re^{101}$, Delete some features (or) Regularization.

1.1.8 Classification

The output value 'y' is **discrete value.**
The algorithm used is **logistic regression.**

1.1.9 Logistic Regression Model

We want $0 \leq h_\theta(x) \leq 1$.

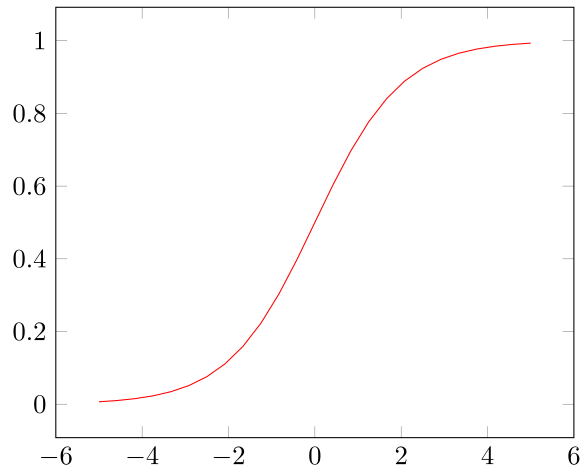
$$h_\theta(x) = g(\theta^\top x) \tag{1.16}$$

$$g(z) = \frac{1}{1+e^{-z}} \tag{1.17}$$

$$h_\theta(x) = g(\theta^\top x) \tag{1.18}$$

$$= \frac{1}{1+e^{-\theta^\top x}} \tag{1.19}$$

The above g(z) is called sigmoid function (or) logistic function.
Graph,



$$g(z) \geq 0.5,\ when\ z \geq 0. \tag{1.20}$$

$$h_\theta(x) = g(\theta^\top x) \geq 0.5,\ when\ \theta^\top x \geq 0. \tag{1.21}$$

1.1.10 Interpretation of hypothesis output for Logistic Regression

$$h_\theta(x) = P(y=1|x;\theta) \tag{1.22}$$

$$P(y=0|x;\theta) + P(y=1|x;\theta) = 1 \tag{1.23}$$

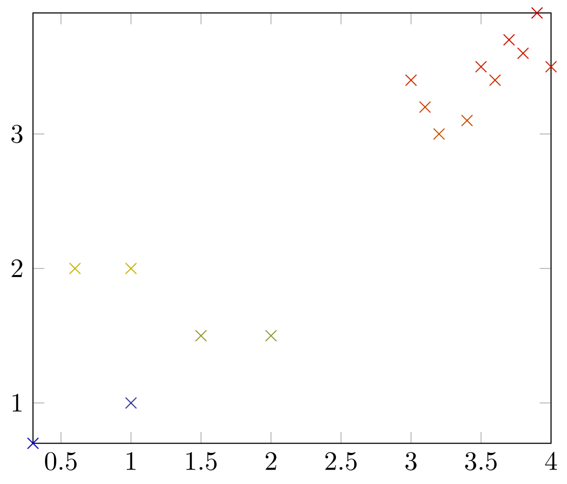
$$P(y=0|x;\theta) = 1 - P(y=1|x;\theta) \tag{1.24}$$

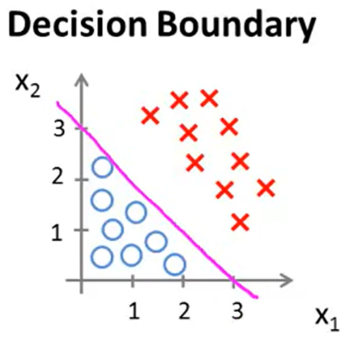
$$= 1 - h_\theta(x) \tag{1.25}$$

$$\tag{1.26}$$

y = 0 (or) 1.

Decision boundary,





For, the above diagram decision boundary will be a line separating the two output values of y(y=0 (or) y=1).

$$h_\theta(x) \geq 0.5 \rightarrow y = 1. \tag{1.27}$$
$$h_\theta(x) < 0.5 \rightarrow y = 0. \tag{1.28}$$
$$h_\theta(x) = g(\theta^\top x) \tag{1.29}$$
$$= g(\theta_0 + \theta_1 x_1 + \theta_2 x_2) \tag{1.30}$$

$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix}$ $\qquad x = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}$

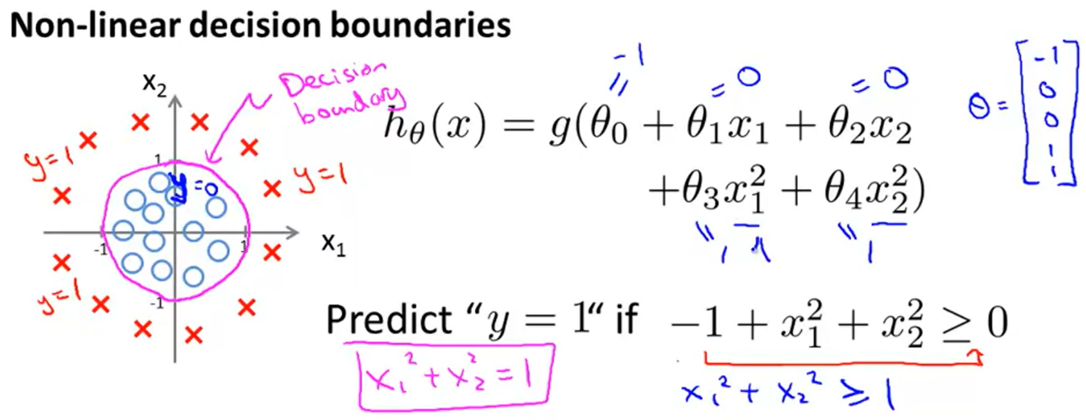
Example,

Let,

$$\theta = \begin{bmatrix} -3 \\ 1 \\ 1 \end{bmatrix}$$

$$y = 1, if\ \theta^\top x \geq 0 \tag{1.31}$$
$$-3 + x_1 + x_2 \geq 0 \tag{1.32}$$
$$x_1 + x_2 \geq 3,\ \ y = 1 \tag{1.33}$$
$$x_1 + x_2 < 3,\ \ y = 0. \tag{1.34}$$

Non-Linear Decision Boundary,



Non-linear decision boundaries
Decision boundary
x_2
x_1
$y=1$
$y=1$
$y=1$
$y=0$
1
-1
1
-1
$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2)$
-1
=0
=0
=1
=1
$\theta = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$
Predict "$y = 1$" if $-1 + x_1^2 + x_2^2 \geq 0$
$x_1^2 + x_2^2 = 1$
$x_1^2 + x_2^2 \geq 1$

So, in the above non-linear classification, the **decision boundary is a circle** of radius of 1unit.

$$Inside\ circle,\ \ y = 0 \tag{1.35}$$
$$Outside\ circle,\ \ y = 1. \tag{1.36}$$

1.1.11 Cost function for Logistic Regression

Training set,

x$^{(1)}$	y$^{(1)}$
x$^{(2)}$	y$^{(2)}$
x$^{(3)}$	y$^{(3)}$
$\vdots$	$\vdots$
x$^{(m)}$	y$^{(m)}$

Table 1.5: Training set of m-examples for Logistic Regression

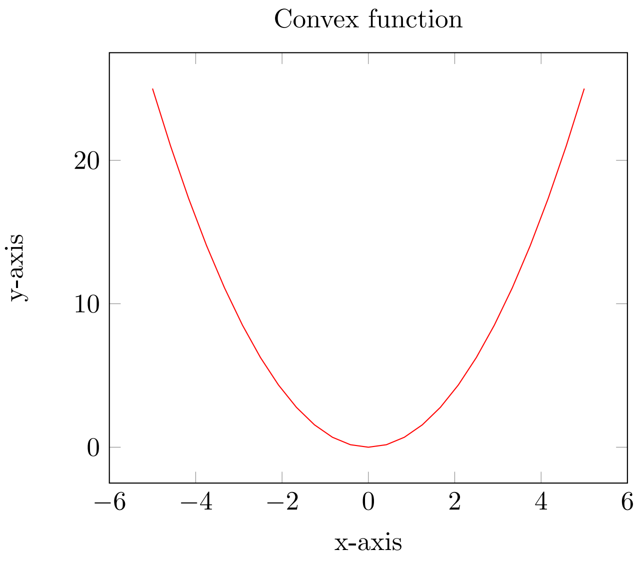
$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \epsilon \ \Re^{n+1} \ - \ n \ features. \ x_0 = 1, \ y \ \epsilon \ 0,1.$$

For linear regression,

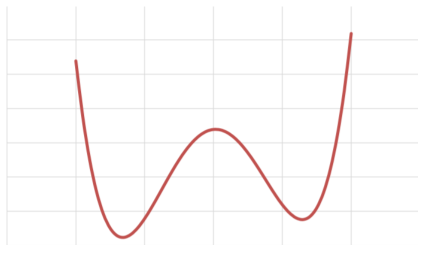
$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} [h_\theta(x^{(i)} - y^{(i)}]^2 \tag{1.37}$$

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} Cost(h_\theta(x^{(i)}, y^{(i)}) \tag{1.38}$$

$$Cost(h_\theta(x^{(i)}), y^{(i)}) = \frac{1}{2}[h_\theta(x^{(i)} - y^{(i)}]^2 \tag{1.39}$$



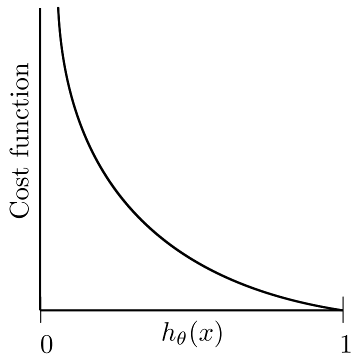
The above graph is a convex.



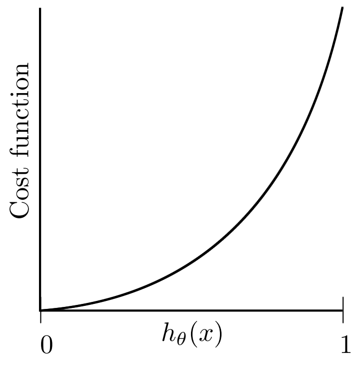
The below graph is a non-convex.
If we use the cost function of linear regression in logistic regression, the the we would get non-convex cost function, because the **hypothesis is** $\frac{1}{1+e^{-\theta^\top x}}$.
For Logistic regression,

$$Cost(h_\theta(x^{(i)}), y^{(i)}) = \begin{cases} -\log(1-h_\theta(x)), & if\ \ y=0 \\ -\log(h_\theta(x)), & if\ \ y=1. \end{cases} \tag{1.40}$$

If y=1,



If y=0,



Simplified Cost function,

$$Cost(h_\theta(x^{(i)}), y^{(i)}) = -(1-y)\log(1-h_\theta(x)) - y\log(h_\theta(x)).\ \forall\ y\ \epsilon\ \{0,1\}. \tag{1.41}$$

$$If\ \ y = 1 : Cost(h_\theta(x), y) = -\log(h_\theta(x)). \tag{1.42}$$

$$If\ \ y = 0 : Cost(h_\theta(x), y) = -\log(1-h_\theta(x)). \tag{1.43}$$

$$Costfunction = J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}, y^{(i)}) \tag{1.44}$$

$$= \frac{1}{m}[-\sum_{i=1}^{m}(1-y^{(i)})\log(1-h_\theta(x^{(i)})) + y^{(i)}\log(h_\theta(x^{(i)}))] \tag{1.45}$$

1.1.12 Gradient Descent for Logistic Regression

$$\text{Repeat until convergence (minimum)}\begin{cases}\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta)\end{cases} \tag{1.46}$$

$$\frac{\partial}{\partial\theta_j}J(\theta) = \frac{1}{m}\sum_{i=1}^{m}[h_\theta(x^{(i)}) - y^{(i)}]x_j^{(i)} \tag{1.47}$$

1.1.13 Optimization algorithm

1. Gradient descent.
2. Conjugate gradient.
3. BFGS.
4. L - BFGS.

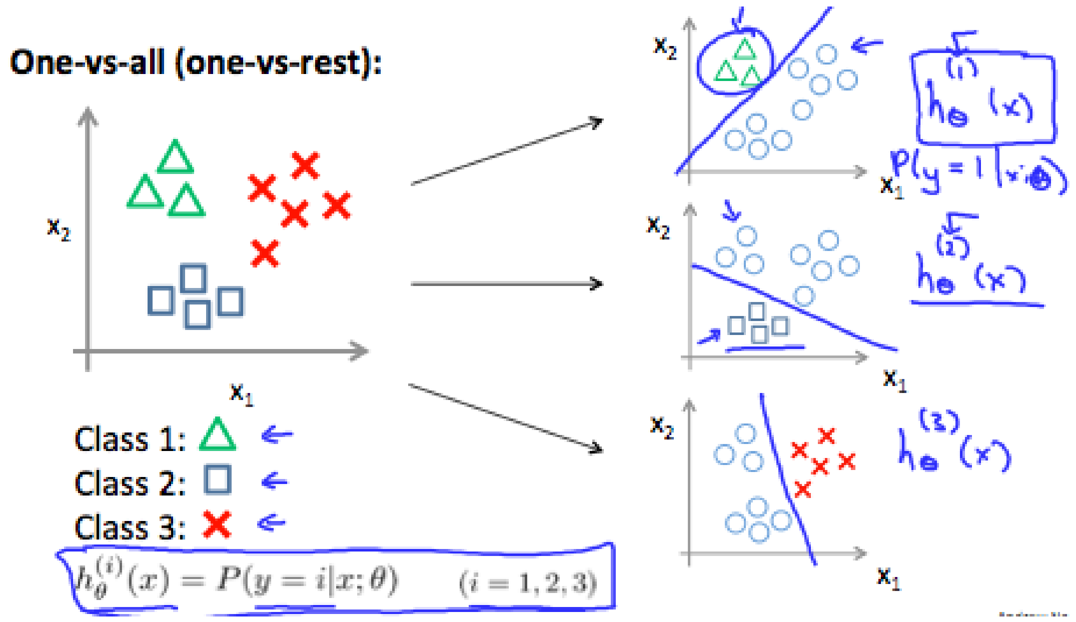
These are the 4 algorithms to minimise **cost function.**
Advantages of the **last three advanced optimization algorithm.**

- No need to manually pick α.
- Often faster than Gradient descent.
- They themselves choose α, for faster convergence.

1.1.14 Multiclass Classification : One-vs-All

$$
\begin{gathered}
y \in \{0, 1...n\} \\
h_\theta^{(0)}(x) = P(y = 0|x;\theta) \\
h_\theta^{(1)}(x) = P(y = 1|x;\theta) \\
\vdots \\
h_\theta^{(n)}(x) = P(y = n|x;\theta) \\
\text{prediction} = \max_i(h_\theta^{(i)}(x))
\end{gathered}
$$

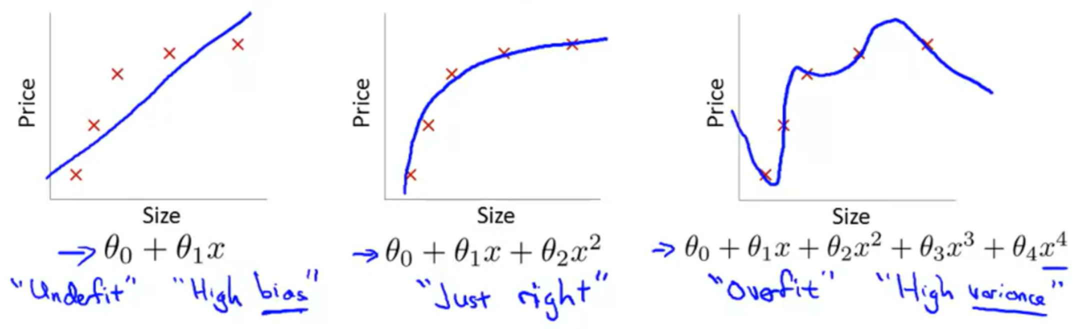
To summarize,



1. Train a logistic regression classifier $h_\theta(x)$ for each class to predict the probability that y=i.
2. To make a prediction on a new x, pick the class that maximizes $h_\theta(x)$

1.1.15 Problem of Overfitting,

Example: Linear regression (housing prices)



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 (predict prices on new examples).

Similar for Logistic Regression.

Addressing Overfitting,

1. Reduce number of features.
   - Manually select which features to keep.
   - Model selection algorithm.
2. Regularization
   - Keep all features, but reduce magnitude/values of parameters $_j$.
   - Works well when we have a lot of features, each of which contributes a bit to predict $y^{(i)}$.

1.1.16 Regularization

Small values for parameters $\theta_0, \theta_1, \theta_2, ..., \theta_n$.

1. Simpler hypothesis.
2. Less prone to overfitting.

Regularized Cost function,

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left[[h_\theta(x^{(i)} - y^{(i)}]^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right] \tag{1.48}$$

If 'λ' is extremely large, then the cost function will become underfitting (doesn't fit to our training data).

Repeat {

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

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

}

The term $\frac{\lambda}{m}\theta_j$ performs our regularization. With some manipulation our update rule can also be represented as:

$$\theta_j := \theta_j(1 - \alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \tag{1.49}$$

The first term in the above equation, $1 - \alpha\frac{\lambda}{m}$ will always be less than 1. Intuitively you can see it as reducing the value of θ_j by some amount on every update. Notice that the second term is now exactly the same as it was before.
Normal equation after regularization,

$$\theta = (X^\top X) + \lambda.L^{-1} X^\top Y \tag{1.50}$$

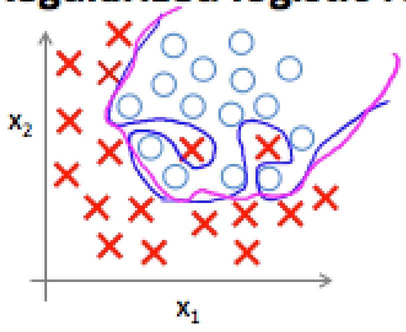
$$where\ L = \begin{bmatrix} 0 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix}$$

1.1.17 Regularized Logistic Regression

Cost function,

$$J(\theta) = \frac{1}{m}\left[-\sum_{i=1}^{m}(1 - y^{(i)})\log(1 - h_\theta(x^{(i)})) + y^{(i)}\log(h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{1.51}$$

Regularized logistic regression.



$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^2 x_2 + \theta_4 x_1^2 x_2^2 + \theta_5 x_1^2 x_2^3 + \dots)$$

The second sum, $\sum_{j=1}^{n} \theta_j^2$ means to explicitly exclude the bias term, θ_0. I.e. the vector is indexed from 0 to n (holding n+1 values, θ_0 through θ_n), and this sum explicitly skips θ_0, by running from 1 to n, skipping 0. Thus, when computing the equation, we should continuously update the two following equations:

Repeat {

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

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

}

1.2 Neural Networks

1.2.1 Model representation I

Let's examine how we will represent a hypothesis function using neural networks. At a very simple level, neurons are basically computational units that take inputs (dendrites) as electrical inputs (called **"spikes"**) that are channeled to outputs (axons). In our model, our dendrites are like the input features $x_1 \cdots x_n$, and the output is the result of our hypothesis function. In this model our x_0 input node is sometimes called the **"bias unit".** It is always equal to 1. In neural networks, we use the same logistic function as in classification, $\frac{1}{1+e^{-\theta^T x}}$, yet we sometimes call it a sigmoid (logistic) activation function. In this situation, our **"theta" parameters** are sometimes called **"weights".**
A simple representation looks like :

$$\left[\; x_0 x_1 x_2 \; \right] \rightarrow [\,] \rightarrow h_\theta(x)$$

Our input nodes (layer 1), also known as the **"input layer"**, go into another node (layer 2), which finally outputs the hypothesis function, known as the **"output layer".**
We can have intermediate layers of nodes between the input and output layers called the **"hidden layers."**
In this example, we label these intermediate or **"hidden"** layer nodes $a_0^2, \cdots, a_n^2 a$ and call them **"activation units."**

1. $a_i^{(j)}$ = "activation" of unit i in layer j
2. $\Theta^{(j)}$ = matrix of weights controlling function mapping from layer j to layer j+1

If we had one hidden layer, it would look like :

$$\left[\ x_0 x_1 x_2 x_3\ \right] \rightarrow \left[a_1^{(2)} a_2^{(2)} a_3^{(2)}\right] \rightarrow h_\theta(x)$$

The values for each of the **"activation"** nodes is obtained as follows :

$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3) \quad (1.52)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3) \quad (1.53)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3) \quad (1.54)$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)}) \quad (1.55)$$

This is saying that we compute our activation nodes by using a 3×4 matrix of parameters. We apply each row of the parameters to our inputs to obtain the value for one activation node. Our hypothesis output is the logistic function applied to the sum of the values of our activation nodes, which have been multiplied by yet another parameter matrix $\Theta^{(2)}$ containing the weights for our second layer of nodes.
Each layer gets its own matrix of weights, $\Theta^{(j)}$.
The dimensions of these matrices of weights is determined as follows :
If network has s_j units in layer j and s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$.
The +1 comes from the addition in $\Theta^{(j)}$ of the **"bias nodes,"** x_0 and $\Theta_0^{(j)}$. In other words the output nodes will not include the bias nodes while the inputs will.

1.2.2 Model representation II

we'll do a vectorized implementation of the above functions. We're going to define a new variable $z_k^{(j)}$ that encompasses the parameters inside our g function.

$$a_1^{(2)} = g(z_1^{(2)}) \tag{1.56}$$
$$a_2^{(2)} = g(z_2^{(2)}) \tag{1.57}$$
$$a_3^{(2)} = g(z_3^{(2)}) \tag{1.58}$$

In other words, for layer j=2 and node k, the variable z will be :

$$z_k^{(2)} = \Theta_{k,0}^{(1)} x_0 + \Theta_{k,1}^{(1)} x_1 + \cdots + \Theta_{k,n}^{(1)} x_n \tag{1.59}$$

The vector representation of x and z^j is :

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \qquad z^{(j)} = \begin{bmatrix} z_1^{(j)} \\ z_2^{(j)} \\ \vdots \\ z_n^{(j)} \end{bmatrix}$$

Setting x = $a^{(1)}$, we can rewrite the equation as :

$$z^{(j)} = \theta^{(j-1)} a^{(j-1)} \tag{1.60}$$

We are multiplying our matrix $\Theta^{(j-1)}$ with dimensions $s_j \times$ (n+1)(where s_j is the number of our activation nodes) by our vector $a^{(j-1)}$ with height (n+1). This gives us our vector $z^{(j)}$ with height s_j. Now we can get a vector of our activation nodes for layer j as follows :

$$a^{(j)} = g(z^{(j)})a \tag{1.61}$$

Where our function g can be applied element-wise to our vector $z^{(j)}$
We can then add a bias unit (equal to 1) to layer j after we have computed $a^{(j)}$. This will be element $a_0^{(j)}$ and will be equal to 1. To compute our final hypothesis, let's first compute another z vector :

$$z^{(j+1)} = \Theta^{(j)} a^{(j)} \tag{1.62}$$

We get this final z vector by multiplying the next theta matrix after $\Theta^{(j-1)}$ with the values of all the activation nodes we just got. This last theta matrix $\Theta^{(j)}$ will have only one row which is multiplied by one column $a^{(j)}$ so that our result is a single number. We then get our final result with :

$$h_\Theta(x) = a^{(j+1)} = g(z^{(j+1)}) \tag{1.63}$$

Notice that in this last step, between layer j and layer j+1, we are doing exactly the same thing as we did in logistic regression. Adding all these intermediate layers in neural networks allows us to more elegantly produce interesting and more complex non-linear hypothesis.

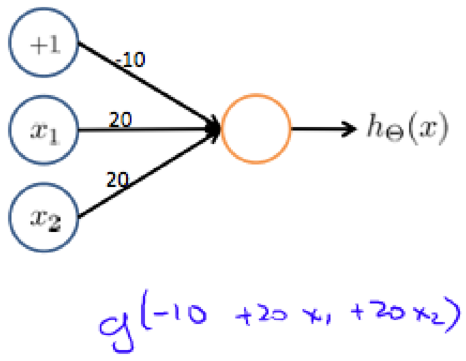
Examples,

A simple example of applying neural networks is by predicting x_1 AND x_2, which is the logical 'and' operator and is only true if both x_1 and x_2 are 1.

$$
\begin{aligned}
h_\Theta(x) &= g(-30 + 20x_1 + 20x_2) \\
&x_1 = 0 \;\; and \;\; x_2 = 0 \;\; then \;\; g(-30) \approx 0 \\
&x_1 = 0 \;\; and \;\; x_2 = 1 \;\; then \;\; g(-10) \approx 0 \\
&x_1 = 1 \;\; and \;\; x_2 = 0 \;\; then \;\; g(-10) \approx 0 \\
&x_1 = 1 \;\; and \;\; x_2 = 1 \;\; then \;\; g(10) \approx 1
\end{aligned}
$$

So we have constructed one of the fundamental operations in computers by using a small neural network rather than using an actual AND gate. Neural networks can also be used to simulate all the other logical gates. The following is an example of the logical operator 'OR', meaning either x_1 is true or x_2 is true, or both :

Example: OR function



x_1	x_2	$h_\Theta(x)$
0	0	$g(-10) \approx 0$
0	1	$g(10) \approx 1$
1	0	≈ 1
1	1	≈ 1

The $^{(1)}$ matrices for AND, NOR, and OR are :

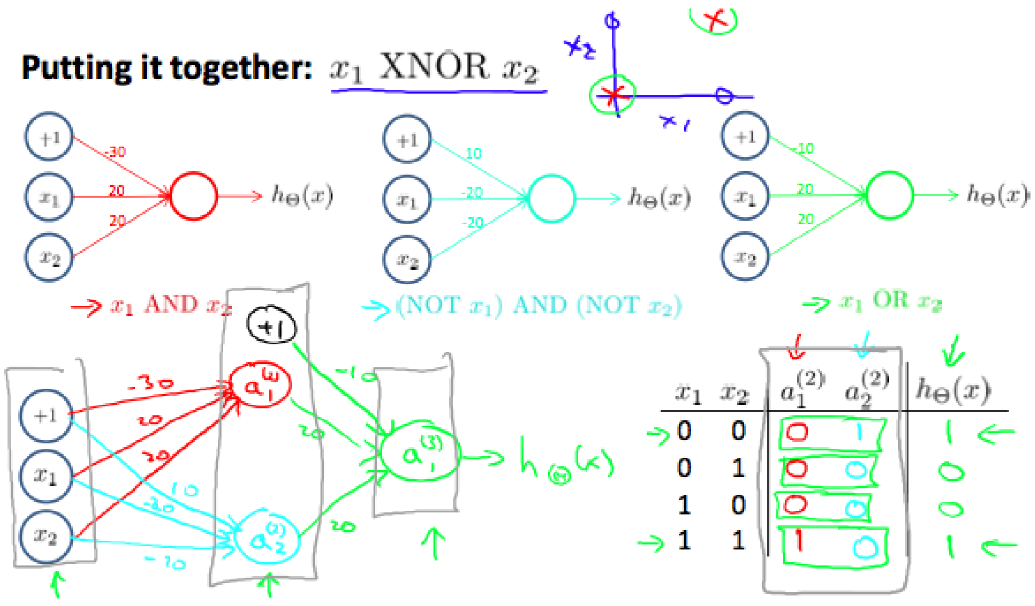
$$
\begin{aligned}
&AND: \\
&\quad \Theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 \end{bmatrix} \\
&NOR: \\
&\quad \Theta^{(1)} = \begin{bmatrix} 10 & -20 & -20 \end{bmatrix} \\
&OR: \\
&\quad \Theta^{(1)} = \begin{bmatrix} -10 & 20 & 20 \end{bmatrix}
\end{aligned}
$$

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).

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \end{bmatrix} \rightarrow \begin{bmatrix} a^{(3)} \end{bmatrix} \rightarrow h_\Theta(x)$$

Let's write out the values for all our nodes :

$$\begin{aligned} a^{(2)} &= g(\Theta^{(1)} \cdot x) \\ a^{(3)} &= g(\Theta^{(2)} \cdot a^{(2)}) \\ h_\Theta(x) &= a^{(3)} \end{aligned}$$

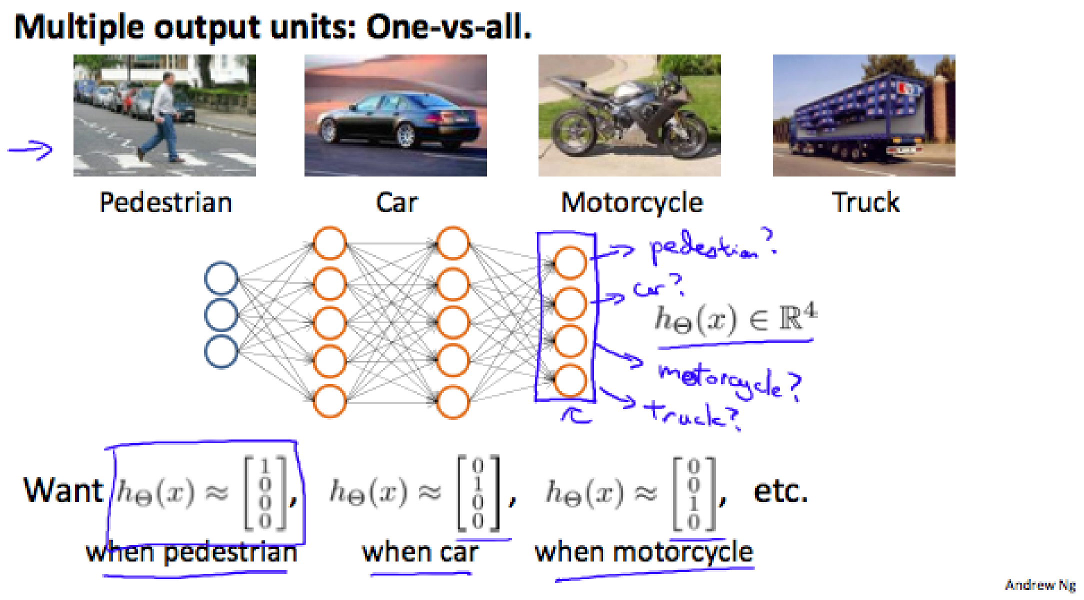


Putting it together: x_1 XNOR x_2
x_1 AND x_2
(NOT x_1) AND (NOT x_2)
x_1 OR x_2
-30
20
20
10
-20
-20
-10
20
20
$h_\Theta(x)$

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

1.2.3 Multiclass Classification

To classify data into multiple classes, we let our hypothesis function return a vector of values. Say we wanted to classify our data into one of four categories. We will use the following example to see how this classification is done. This algorithm takes as input an image and classifies it accordingly :



Multiple output units: One-vs-all.

Want $\boxed{h_\Theta(x) \approx \begin{bmatrix}1\\0\\0\\0\end{bmatrix}}$, $h_\Theta(x) \approx \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $h_\Theta(x) \approx \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, etc.

when pedestrian   when car   when motorcycle

Andrew Ng

We can define our set of resulting classes as y :

$$y^{(i)} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix},$$

Each $y^{(i)}$ represents a different image corresponding to either a car, pedestrian, truck, or motorcycle. The inner layers, each provide us with some new information which leads to our final hypothesis function. The setup looks like :

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix} \rightarrow \begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \\ \cdots \end{bmatrix} \rightarrow \begin{bmatrix} a_0^{(3)} \\ a_1^{(3)} \\ a_2^{(3)} \\ \cdots \end{bmatrix} \rightarrow \cdots \rightarrow \begin{bmatrix} h_\Theta(x)_1 \\ h_\Theta(x)_2 \\ h_\Theta(x)_3 \\ h_\Theta(x)_4 \end{bmatrix}$$

Our resulting hypothesis for one set of inputs may look like :

$$h_\theta(x) = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}$$

In which case our resulting class is the third one down, or $h_\Theta(x)_3$, which represents the motorcycle.

1.2.4 Cost function for Neural networks

Let's declare some variables.

1. L = total number of layers in the network.
2. s_l = number of units (not counting bias unit) in layer l.
3. K = number of output units/classes.

The cost function for regularized logistic regression,

$$J(\theta) = \frac{1}{m}\left[-\sum_{i=1}^{m}(1-y^{(i)})\log(1-h_\theta(x^{(i)})) + y^{(i)}\log(h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{1.64}$$

The cost function for Neural networks,

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[(1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k) + y_k^{(i)}\log((h_\theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{j,i}^{(l)})^2 \tag{1.65}$$

In the regularization part, after the square 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). As before with logistic regression, we square every term.

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

1.2.5 Backpropagation Algorithm

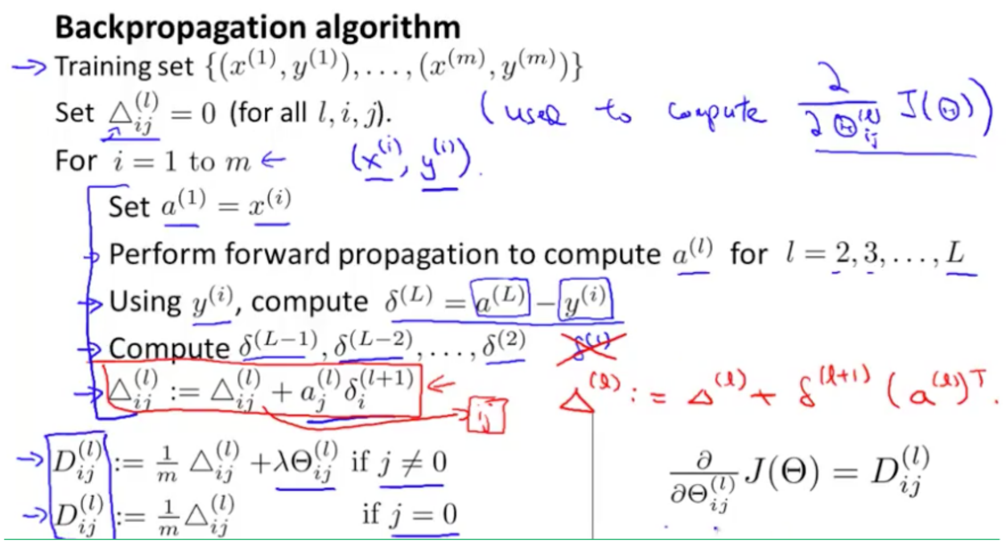
"Backpropagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression. Our goal is to compute :

$$\min_{\Theta} J(\Theta) \tag{1.66}$$

That is, we want to minimize our cost function J using an optimal set of parameters in theta. In this section we'll look at the equations we use to compute the partial derivative of J() :

$$\frac{\partial}{\partial \Theta^{(l)}_{j,i}} J(\Theta) \tag{1.67}$$

Backpropagation algorithm,



Backpropagation algorithm

```
Training set $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$
Set $\triangle^{(l)}_{ij} = 0$ (for all $l, i, j$).   $\left(\text{used to compute } \frac{\partial}{\partial \Theta^{(l)}_{ij}} J(\Theta)\right)$
For $i = 1$ to $m$   $(x^{(i)}, y^{(i)})$.
    Set $a^{(1)} = x^{(i)}$
    Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \dots, L$
    Using $y^{(i)}$, compute $\delta^{(L)} = \boxed{a^{(L)}} - \boxed{y^{(i)}}$
    Compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$   $\cancel{\delta^{(1)}}$
    $\boxed{\triangle^{(l)}_{ij} := \triangle^{(l)}_{ij} + a^{(l)}_j \delta^{(l+1)}_i}$   $\boxed{ij}$
        $\triangle^{(l)} := \triangle^{(l)} + \delta^{(l+1)} (a^{(l)})^T.$
$\boxed{D^{(l)}_{ij}} := \frac{1}{m} \triangle^{(l)}_{ij} + \lambda \Theta^{(l)}_{ij}$   if $j \neq 0$
$\boxed{D^{(l)}_{ij}} := \frac{1}{m} \triangle^{(l)}_{ij}$   if $j = 0$
        $\frac{\partial}{\partial \Theta^{(l)}_{ij}} J(\Theta) = D^{(l)}_{ij}$
```

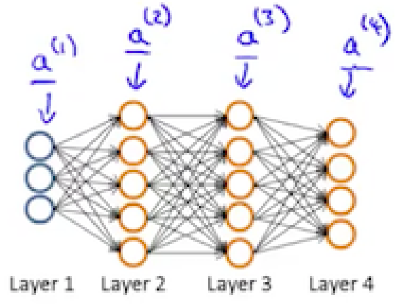
1. Given training set $\{(x^{(1)}, y^{(1)}) \cdots (x^{(m)}, y^{(m)})\}$.
    - Set $\Delta^{(l)}_{i,j} := 0$ for all (l,i,j), (hence you end up having a matrix full of zeros)
2. For training example t = 1 to m :
    - Set $a^{(1)} := x^{(t)}$.
    - Perform forward propagation to compute $a^{(l)}$ for l=2,3,...,L.
    - Using $y^{(t)}$, compute $\delta^{(L)} = a^{(L)} - y^{(t)}$. Where L is our total number of layers and $a^{(L)}$ is the vector of outputs of the activation units for the last layer. So our **"error values"** for the last layer are simply the differences of our actual results in the last layer and the correct outputs in y. To get the delta values of the layers before the last layer, we can use an equation that steps us back from right to left :

Gradient computation

Given one training example ($\underline{x, y}$):

Forward propagation:

$$\underline{a^{(1)}} = \underline{x}$$
$$\rightarrow z^{(2)} = \Theta^{(1)} a^{(1)}$$
$$\rightarrow a^{(2)} = g(z^{(2)}) \quad (\underline{\text{add } a_0^{(2)}})$$
$$\rightarrow z^{(3)} = \Theta^{(2)} a^{(2)}$$
$$\rightarrow a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$
$$\rightarrow z^{(4)} = \Theta^{(3)} a^{(3)}$$
$$\rightarrow a^{(4)} = \underline{h_\Theta(x)} = g(z^{(4)})$$



- Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ using $\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)})\ .*\ a^{(l)}\ .*\ (1 - a^{(l)})$.
  The delta values of layer l are calculated by multiplying the delta values in the next layer with the theta matrix of layer l. We then element-wise multiply that with a function called g', or g-prime, which is the derivative of the activation function g evaluated with the input values given by $z^{(l)}$.
  The g-prime derivative terms can also be written out as :

$$g'(z^{(l)}) = a^{(l)}\ .*\ (1 - a^{(l)}) \tag{1.68}$$

- $\Delta^{(l)}_{i,j} := \Delta^{(l)}_{i,j} + a^{(l)}_j \delta^{(l+1)}_i$ (or) with vectorization, $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$.
  Hence we update our new Δ matrix.
  (a) $D^{(l)}_{i,j} := \frac{1}{m}(\Delta^{(l)}_{i,j} + \lambda\Theta^{(l)}_{i,j})$, if j$\neq$ 0.
  $\mathrm{D}^{(l)}_{i,j} := \frac{1}{m}(\Delta^{(l)}_{i,j}$, if j = 0.
  The capital-delta matrix D is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative. Thus we get $\frac{\partial}{\partial\Theta^{(l)}_{ij}}J(\Theta) = D^{(l)}_{ij}$.

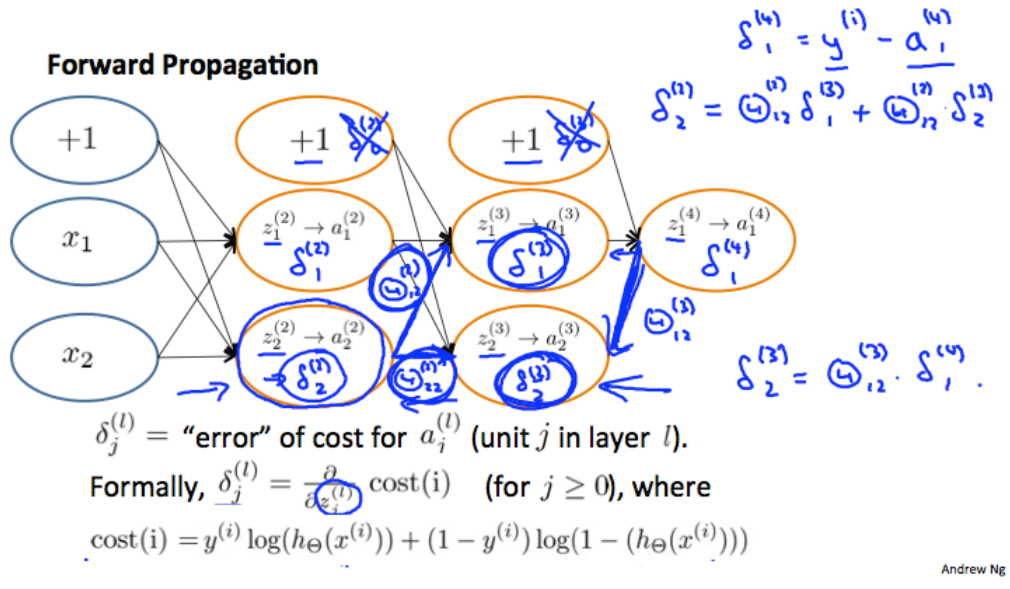
If we consider simple non-multiclass classification (k = 1) and disregard regularization, the cost is computed with :

$$Cost(t) = (1 - y^{(t)})\log(1 - h_\theta(x^{(t)})) + y^{(t)}\log(h_\theta(x^{(t)})) \qquad (1.69)$$

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

$$\delta^{(l)}_j = \frac{\partial}{\partial z^{(l)}_j} cost(t) \qquad (1.70)$$

Our derivative is the slope of a line tangent to the cost function, so the steeper the slope the more incorrect we are. Let us consider the following neural network below and see how we could calculate some $\delta^{(l)}_j$:


Forward Propagation
$\delta^{(4)}_1 = y^{(i)} - a^{(4)}_1$
$\delta^{(2)}_2 = \Theta^{(2)}_{12}\delta^{(3)}_1 + \Theta^{(2)}_{22}\delta^{(3)}_2$
$\delta^{(3)}_2 = \Theta^{(3)}_{12} \cdot \delta^{(4)}_1$
+1
x_1
x_2
$z^{(2)}_1 \to a^{(2)}_1$
$z^{(2)}_2 \to a^{(2)}_2$
$z^{(3)}_1 \to a^{(3)}_1$
$z^{(3)}_2 \to a^{(3)}_2$
$z^{(4)}_1 \to a^{(4)}_1$
$\delta^{(l)}_j$ = "error" of cost for $a^{(l)}_j$ (unit j in layer l).
Formally, $\delta^{(l)}_j = \frac{\partial}{\partial z^{(l)}_j}\mathrm{cost}(i)$ (for $j \geq 0$), where
$\mathrm{cost(i)} = y^{(i)}\log(h_\Theta(x^{(i)})) + (1 - y^{(i)})\log(1 - (h_\Theta(x^{(i)})))$
Andrew Ng

In the image above, to calculate $\delta_2^{(2)}$, we multiply the weights $\Theta_{12}^{(2)}$ and $\Theta_{22}^{(2)}$ by their respective δ values found to the right of each edge. So we get $\delta_2^{(2)} = \Theta_{12}^{(2)} * \delta_1^{(3)} + \Theta_{22}^{(2)} * \delta_2^{(3)}$. To calculate every single possible $\delta_j^{(l)}$, we could start from the right of our diagram. We can think of our edges as our Θ_{ij}. Going from right to left, to calculate the value of $\delta_j^{(l)}$, you can just take the over all sum of each weight times the δ it is coming from. Hence, another example would be $\delta_2^{(3)} = \Theta_{12}^{(3)} * \delta_1^{(4)}$.

1.2.6 Gradient Checking

Gradient checking will assure that our backpropagation works as intended. We can approximate the derivative of our cost function with :

$$\frac{\partial}{\partial\theta} J(\Theta) \approx \frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon} \tag{1.71}$$

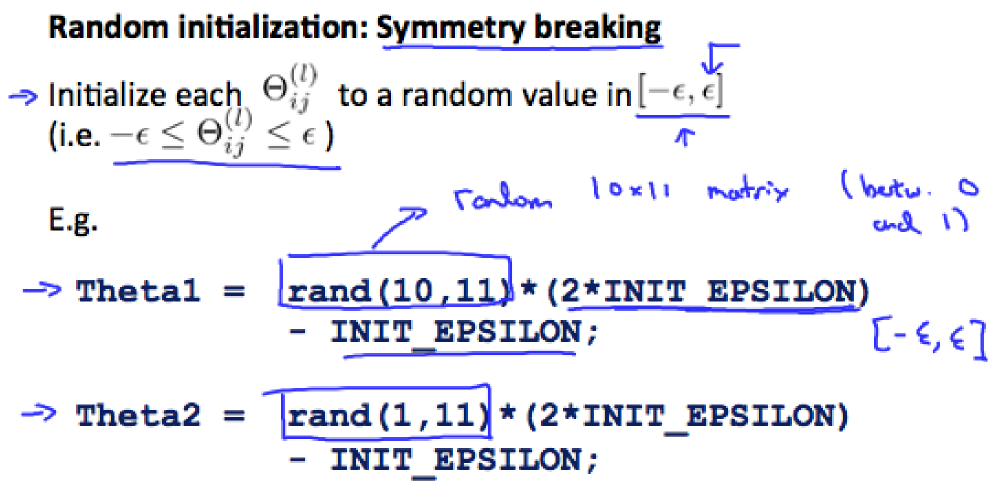
With multiple theta matrices, we can approximate the derivative with respect to Θ_j as follows :

$$\frac{\partial}{\partial\theta_j} J(\Theta) \approx \frac{J(\Theta_1, ..., \Theta_j + \epsilon, ..., \Theta_n) - J(\Theta_1, ..., \Theta_j - \epsilon, ..., \Theta_n)}{2\epsilon} \tag{1.72}$$

A small value for ϵ (epsilon) such as $\epsilon = 10^{-4}$, guarantees that the math works out properly. If the value for ϵ is too small, we can end up with numerical problems.
We previously saw how to calculate the deltaVector. So once we compute our gradApprox vector, we can check that gradApprox $\approx$ deltaVector.
Once you have verified once that your backpropagation algorithm is correct, you don't need to compute gradApprox again. The code to compute gradApprox can be very slow.

1.2.7 Random Intialization

Initializing all theta weights to zero does not work with neural networks. When we backpropagate, all nodes will update to the same value repeatedly. Instead we can randomly initialize our weights for our Θ matrices using the following method :



Random initialization: Symmetry breaking

→ Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$
(i.e. $-\epsilon \le \Theta_{ij}^{(l)} \le \epsilon$)

E.g.

random 10×11 matrix (betw. 0 and 1)

```
→ Theta1 = rand(10,11)*(2*INIT_EPSILON)
           - INIT_EPSILON;
```

$[-\epsilon, \epsilon]$

```
→ Theta2 = rand(1,11)*(2*INIT_EPSILON)
           - INIT_EPSILON;
```

Hence, we initialize each $\Theta_{ij}^{(l)}$ to a random value between $[-\epsilon, \epsilon]$. Using the above formula guarantees that we get the desired bound.

The epsilon used above is unrelated to the epsilon from Gradient Checking.

1.2.8 Choosing Neural network

1. Pick a network architecture.
2. Choose the layout of your neural network.
3. Including how many hidden units in each layer and how many layers in total you want to have.
   - Number of input units = dimension of features $x^{(i)}$.
   - Number of output units = number of classes.
   - Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units).
   - **Defaults** : 1 hidden layer. If you have more than 1 hidden layer, then it is recommended that you have the same number of units in every hidden layer.

1.2.9 Training a Neural Network

1. Randomly initialize the weights.
2. Implement forward propagation to get $h_\Theta(x^{(i)})$.

3. Implement the cost function.
4. Implement backpropagation to compute partial derivatives.
5. Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.
6. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.

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