

[Previous](#) [Next](#) [Index](#)

## Matrices - overview

- Rectangular array of numbers written between square brackets
  - 2D array
  - Named as capital letters (A,B,X,Y)
- Dimension of a matrix are [Rows x Columns]
  - Start at top left
  - To bottom left
  - To bottom right
  - $R^{[r \times c]}$  means a matrix which has r rows and c columns

$$A = \begin{bmatrix} 1402 & 191 \\ 1371 & 821 \\ 949 & 1437 \\ 147 & 1448 \end{bmatrix}$$

- Is a [4 x 2] matrix

- Matrix elements
  - $A_{(i,j)}$  = entry in  $i^{\text{th}}$  row and  $j^{\text{th}}$  column

Handwritten diagram showing matrix  $A$  with row and column indices and corresponding element values:

$$A = \begin{bmatrix} 1402 & 191 \\ 1371 & 821 \\ 949 & 1437 \\ 147 & 1448 \end{bmatrix}$$

Indices and values shown:

- $A_{11} = 1402$  (blue)
- $A_{12} = 191$  (red)
- $A_{32} = 1437$  (pink)
- $A_{41} = 147$  (cyan)

- Provides a way to organize, index and access a lot of data

## Vectors - overview

- Is an n by 1 matrix
  - Usually referred to as a lower case letter
  - n rows
  - 1 column
  - e.g.

$$y = \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}$$

- Is a 4 dimensional vector
  - Refer to this as a vector  $R_4$
- Vector elements
  - $v_i = i^{\text{th}}$  element of the vector
  - Vectors can be 0-indexed (C++) or 1-indexed (MATLAB)
  - In math 1-indexed is most common
    - But in machine learning 0-index is useful
  - Normally assume using 1-index vectors, but be aware sometimes these will (explicitly) be 0 index ones

## Matrix manipulation

- **Addition**
  - Add up elements one at a time
  - Can only add matrices of the *same dimensions*
    - Creates a new matrix of the same dimensions of the ones added

$$\begin{bmatrix} 1 & 0 \\ 2 & 5 \\ 3 & 1 \end{bmatrix} + \begin{bmatrix} 4 & 0.5 \\ 2 & 5 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 0.5 \\ 4 & 10 \\ 3 & 2 \end{bmatrix}$$

- **Multiplication by scalar**

- Scalar = real number
- Multiply each element by the scalar
- Generates a matrix of the same size as the original matrix

$$3 \times \begin{bmatrix} 1 & 0 \\ 2 & 5 \\ 3 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 0 \\ 6 & 15 \\ 9 & 3 \end{bmatrix}$$

- **Division by a scalar**

- Same as multiplying a matrix by  $1/4$
- Each element is divided by the scalar

- **Combination of operands**

- Evaluate multiplications first

$$3 \times \begin{bmatrix} 1 \\ 4 \\ 2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 5 \end{bmatrix} - \begin{bmatrix} 3 \\ 0 \\ 2 \end{bmatrix} / 3$$

- **Matrix by vector multiplication**

- $[3 \times 2]$  matrix \*  $[2 \times 1]$  vector
  - New matrix is  $[3 \times 1]$ 
    - More generally if  $[a \times b] * [b \times c]$ 
      - Then new matrix is  $[a \times c]$
  - How do you do it?
    - Take the two vector numbers and multiply them with the first row of the matrix
      - Then add results together - this number is the first number in the new vector
    - The multiply second row by vector and add the results together
    - Then multiply final row by vector and add them together

$$\begin{bmatrix} 1 & 3 \\ 4 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \end{bmatrix} = \begin{bmatrix} 16 \\ 4 \\ 7 \end{bmatrix}$$

$3 \times 2 \quad 2 \times 1$

$$1 \times 1 + 3 \times 5 = 16$$

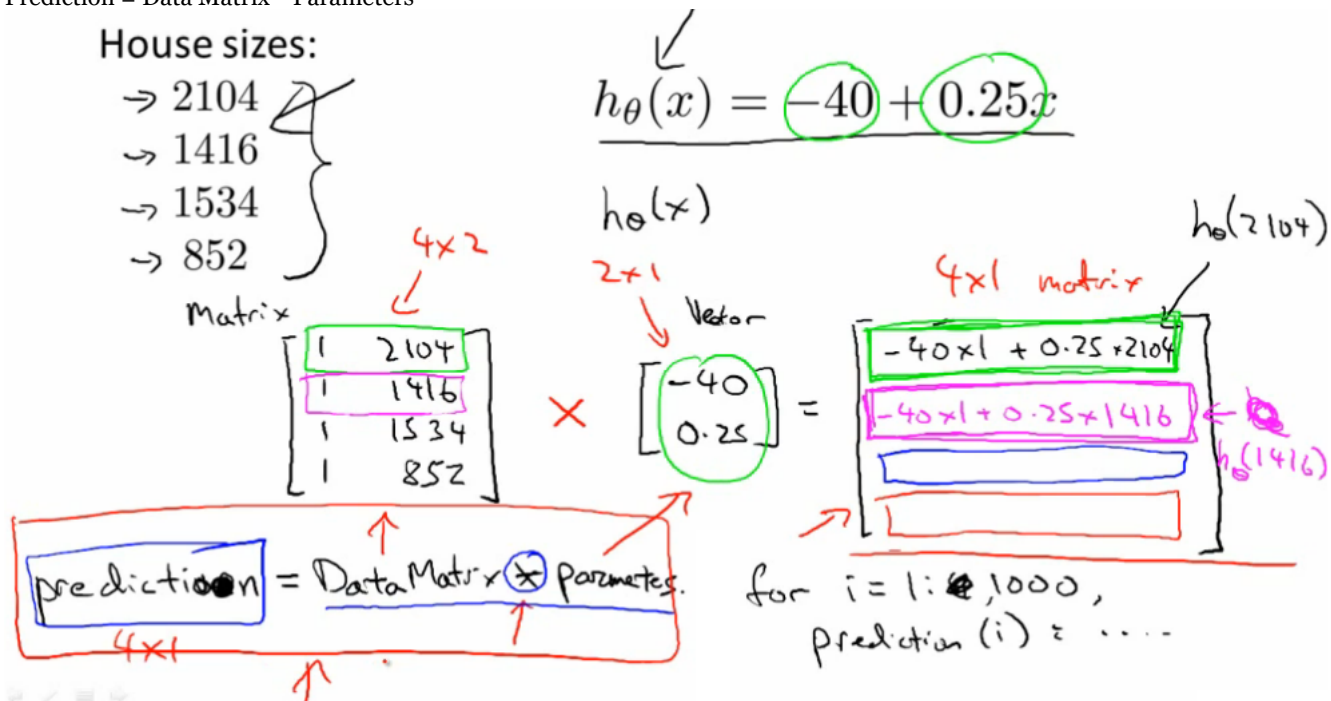
$$4 \times 1 + 0 \times 5 = 4$$

$$2 \times 1 + 1 \times 5 = 7$$

- Detailed explanation

- $A * x = y$ 
  - A is  $m \times n$  matrix
  - x is  $n \times 1$  matrix
  - n must match between vector and matrix
    - i.e. inner dimensions must match
  - Result is an m-dimensional vector

- To get  $y_i$  - multiply A's  $i^{\text{th}}$  row with all the elements of vector  $x$  and add them up
- Neat trick
  - Say we have a data set with four values
  - Say we also have a hypothesis  $h_\theta(x) = -40 + 0.25x$ 
    - Create your data as a matrix which can be multiplied by a vector
    - Have the parameters in a vector which your matrix can be multiplied by
  - Means we can do
    - Prediction = Data Matrix \* Parameters



- Here we add an extra column to the data with 1s - this means our  $\theta_0$  values can be calculated and expressed
- The diagram above shows how this works
  - This can be far more efficient computationally than lots of for loops
  - This is also easier and cleaner to code (assuming you have appropriate libraries to do matrix multiplication)
- **Matrix-matrix multiplication**
  - General idea
    - Step through the second matrix one column at a time
    - Multiply each column vector from second matrix by the entire first matrix, each time generating a vector
    - The final product is these vectors combined (not added or summed, but literally just put together)
  - Details
    - $A \times B = C$ 
      - $A = [m \times n]$
      - $B = [n \times o]$
      - $C = [m \times o]$ 
        - With vector multiplications  $o = 1$
    - Can only multiply matrix where columns in A match rows in B
  - Mechanism
    - Take column 1 of B, treat as a vector
    - Multiply A by that column - generates an  $[m \times 1]$  vector
    - Repeat for each column in B
      - There are  $o$  columns in B, so we get  $o$  columns in C
  - Summary
    - The  $i^{\text{th}}$  column of matrix C is obtained by multiplying A with the  $i^{\text{th}}$  column of B
  - Start with an example
  - $A \times B$

$$\begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 0 & 1 \\ 5 & 2 \end{bmatrix}$$

- Initially
  - Take matrix A and multiply by the first column vector from B
  - Take the matrix A and multiply by the second column vector from B

$$\begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 5 \end{bmatrix} = \begin{bmatrix} 11 \\ 9 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 10 \\ 14 \end{bmatrix}$$

- $2 \times 3$  times  $3 \times 2$  gives you a  $2 \times 2$  matrix

## Implementation/use

- House prices, but now we have three hypothesis and the same data set
- To apply all three hypothesis to all data we can do this efficiently using matrix-matrix multiplication
  - Have
    - Data matrix
    - Parameter matrix
  - Example
    - Four houses, where we want to predict the prize
    - Three competing hypotheses
    - Because our hypothesis are one variable, to make the matrices match up we make our data (houses sizes) vector into a  $4 \times 2$  matrix by adding an extra column of 1s

House sizes:

$$\begin{Bmatrix} 2104 \\ 1416 \\ 1534 \\ 852 \end{Bmatrix}$$

Have 3 competing hypotheses:

$$\begin{aligned} 1. & h_{\theta}(x) = -40 + 0.25x \\ 2. & h_{\theta}(x) = 200 + 0.1x \\ 3. & h_{\theta}(x) = -150 + 0.4x \end{aligned}$$

Matrix

$$\begin{bmatrix} 1 & 2104 \\ 1 & 1416 \\ 1 & 1534 \\ 1 & 852 \end{bmatrix}$$

Matrix

$$\begin{bmatrix} -40 & 200 & -150 \\ 0.25 & 0.1 & 0.4 \end{bmatrix}$$

$$\begin{bmatrix} 486 & 410 & 692 \\ 314 & 342 & 416 \\ 344 & 353 & 464 \\ 173 & 285 & 191 \end{bmatrix}$$

Prediction of first  $h_{\theta}$

Predictions of 2nd  $h_{\theta}$

- What does this mean
  - Can quickly apply three hypotheses at once, making 12 predictions
  - Lots of good linear algebra libraries to do this kind of thing very efficiently

## Matrix multiplication properties

- Can pack a lot into one operation
  - However, should be careful of how you use those operations
  - Some interesting properties
- **Commutativity**
  - When working with raw numbers/scalars multiplication is commutative
    - $3 * 5 == 5 * 3$
  - This is not true for matrix
    - $A \times B \neq B \times A$
    - **Matrix multiplication is not commutative**
- **Associativity**
  - $3 \times 5 \times 2 == 3 \times 10 = 15 \times 2$ 
    - Associative property
  - **Matrix multiplications is associative**
    - $A \times (B \times C) == (A \times B) \times C$
- **Identity matrix**
  - 1 is the identity for any scalar
    - i.e.  $1 \times z = z$

- for any real number
- In matrices we have an identity matrix called  $I$ 
  - Sometimes called  $I_{\{n \times n\}}$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$2 \times 2$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$3 \times 3$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$4 \times 4$

- See some identity matrices above
  - Different identity matrix for each set of dimensions
  - Has
    - 1s along the diagonals
    - 0s everywhere else
  - 1x1 matrix is just "1"
- Has the property that any matrix A which can be multiplied by an identity matrix gives you matrix A back
  - So if A is  $[m \times n]$  then
    - $A * I$ 
      - $I = n \times n$
    - $I * A$ 
      - $I = m \times m$
    - (To make inside dimensions match to allow multiplication)
- Identity matrix dimensions are implicit
- Remember that matrices are not commutative  $AB \neq BA$ 
  - Except when B is the identity matrix
  - Then  $AB == BA$

## Inverse and transpose operations

- **Matrix inverse**
    - How does the concept of "the inverse" relate to real numbers?
      - 1 = "identity element" (as mentioned above)
        - Each number has an inverse
          - This is the number you multiply a number by to get the identity element
          - i.e. if you have x,  $x * 1/x = 1$
      - e.g. given the number 3
        - $3 * 3^{-1} = 1$  (the identity number/matrix)
      - In the space of real numbers **not everything has an inverse**
        - e.g. 0 does not have an inverse
    - What is the inverse of a matrix
      - If A is an  $m \times m$  matrix, then  $A \text{ inverse} = A^{-1}$
      - So  $A * A^{-1} = I$
      - Only matrices which are  $m \times m$  have inverses
        - Square matrices only!
    - Example
      - 2 x 2 matrix
- $$\begin{bmatrix} 3 & 4 \\ 2 & 16 \end{bmatrix} \begin{bmatrix} 0.4 & -0.1 \\ -0.05 & 0.075 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I_{2 \times 2}$$

$A^{-1} A$
- How did you find the inverse
      - Turns out that you can sometimes do it by hand, although this is very hard
      - Numerical software for computing a matrices inverse
        - Lots of open source libraries
  - If A is all zeros then there is no inverse matrix
    - Some others don't, intuition should be matrices that don't have an inverse are a singular matrix or a degenerate matrix (i.e. when it's too close to 0)
    - So if all the values of a matrix reach zero, this can be described as reaching singularity
- **Matrix transpose**
  - Have matrix A (which is  $[n \times m]$ ) how do you change it to become  $[m \times n]$  while keeping the same values
    - i.e. swap rows and columns!
  - How you do it;
    - Take first row of A - becomes 1st column of  $A^T$
    - Second row of A - becomes 2nd column...
  - A is an  $m \times n$  matrix
    - B is a transpose of A
    - Then B is an  $n \times m$  matrix

$$\blacksquare A_{(i,j)} = B_{(j,i)}$$

6/14

$$\underline{A} = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 5 & 9 \end{bmatrix} \quad \underline{A}^T = \begin{bmatrix} 1 & 3 \\ 2 & 5 \\ 0 & 9 \end{bmatrix}$$

[Previous](#) [Next](#) [Index](#)

## Linear regression with multiple features

*New version of linear regression with multiple features*

- Multiple variables = multiple features
- In original version we had
  - $X$  = house size, use this to predict
  - $y$  = house price
- If in a new scheme we have more variables (such as number of bedrooms, number floors, age of the home)
  - $x_1, x_2, x_3, x_4$  are the four features
    - $x_1$  - size (feet squared)
    - $x_2$  - Number of bedrooms
    - $x_3$  - Number of floors
    - $x_4$  - Age of home (years)
  - $y$  is the output variable (price)
- More notation
  - $n$ 
    - number of features ( $n = 4$ )
  - $m$ 
    - number of examples (i.e. number of rows in a table)
  - $x^i$ 
    - vector of the input for an example (so a vector of the four parameters for the  $i^{\text{th}}$  input example)
    - $i$  is an index into the training set
    - So
      - $x$  is an  $n$ -dimensional feature vector
      - $x^3$  is, for example, the 3rd house, and contains the four features associated with that house
  - $x_j^i$ 
    - The value of feature  $j$  in the  $i$ th training example
    - So
      - $x_2^3$  is, for example, the number of bedrooms in the third house
- Now we have multiple features
  - What is the form of our hypothesis?
  - Previously our hypothesis took the form;
    - $h_\theta(x) = \theta_0 + \theta_1 x$ 
      - Here we have two parameters ( $\theta_1$  and  $\theta_2$ ) determined by our cost function
      - One variable  $x$
  - Now we have multiple features
    - $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4$
  - For example
    - $h_\theta(x) = 80 + 0.1x_1 + 0.01x_2 + 3x_3 - 2x_4$ 
      - An example of a hypothesis which is trying to predict the price of a house
      - Parameters are still determined through a cost function
  - For convenience of notation,  $x_0 = 1$ 
    - For every example  $i$  you have an additional 0th feature for each example
    - So now your **feature vector** is  $n + 1$  dimensional feature vector indexed from 0
      - This is a column vector called  $x$
      - Each example has a column vector associated with it
      - So let's say we have a new example called "X"
    - **Parameters** are also in a 0 indexed  $n+1$  dimensional vector
      - This is also a column vector called  $\theta$
      - This vector is the same for each example
  - Considering this, hypothesis can be written
    - $h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4$

- If we do
  - $h_{\theta}(x) = \theta^T X$ 
    - $\theta^T$  is an  $[1 \times n+1]$  matrix
    - In other words, because  $\theta$  is a column vector, the transposition operation transforms it into a row vector
    - So before
      - $\theta$  was a matrix  $[n+1 \times 1]$
    - Now
      - $\theta^T$  is a matrix  $[1 \times n+1]$
    - Which means the inner dimensions of  $\theta^T$  and  $X$  match, so they can be multiplied together as
      - $[1 \times n+1] * [n+1 \times 1]$
      - $= h_{\theta}(x)$
      - So, in other words, the transpose of our parameter vector \* an input example  $X$  gives you a predicted hypothesis which is  $[1 \times 1]$  dimensions (i.e. a single value)
  - This  $x_0 = 1$  lets us write this like this
- This is an example of multivariate linear regression

## Gradient descent for multiple variables

- Fitting parameters for the hypothesis with gradient descent
  - Parameters are  $\theta_0$  to  $\theta_n$
  - Instead of thinking about this as  $n$  separate values, think about the parameters as a single vector ( $\theta$ )
    - Where  $\theta$  is  $n+1$  dimensional
- Our cost function is

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

- Similarly, instead of thinking of  $J$  as a function of the  $n+1$  numbers,  $J()$  is just a function of the parameter vector
  - $J(\theta)$

Repeat {

- **Gradient descent**  $\rightarrow \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \dots, \theta_n)$   
 } (simultaneously update for every  $j = 0, \dots, n$ )

- Once again, this is
  - $\theta_j = \theta_j - \text{learning rate } (\alpha) \text{ times the partial derivative of } J(\theta) \text{ with respect to } \theta_j(\dots)$
  - We do this through a **simultaneous update** of every  $\theta_j$  value
- Implementing this algorithm
  - When  $n = 1$

Repeat {

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

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x^{(i)}$$

(simultaneously update  $\theta_0, \theta_1$ ) }



- Above, we have slightly different update rules for  $\theta_0$  and  $\theta_1$ 
  - Actually they're the same, except the end has a previously undefined  $x_0^{(i)}$  as 1, so wasn't shown
- We now have an almost identical rule for multivariate gradient descent

New algorithm ( $n \geq 1$ ):

Repeat {

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

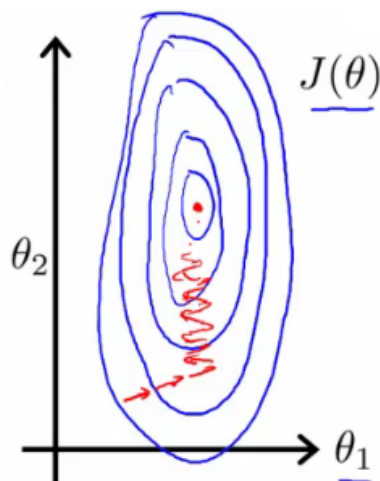
(simultaneously update  $\theta_j$  for  $j = 0, \dots, n$ ) }

$\downarrow \frac{\partial}{\partial \theta_j} J(\theta)$

- What's going on here?
  - We're doing this for each  $j$  (0 until  $n$ ) as a simultaneous update (like when  $n = 1$ )
  - So, we re-set  $\theta_j$  to
    - $\theta_j$  minus the learning rate ( $\alpha$ ) times the partial derivative of the  $J$  vector with respect to  $\theta_j$
    - In non-calculus words, this means that we do
      - Learning rate
      - Times  $1/m$  (makes the maths easier)
      - Times the sum of
        - The hypothesis taking in the variable vector, minus the actual value, times the  $j$ -th value in that variable vector for EACH example
    - It's important to remember that
 
$$\frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} = \frac{\partial}{\partial \theta_j} J(\theta)$$
  - These algorithm are highly similar

## Gradient Decent in practice: 1 Feature Scaling

- Having covered the theory, we now move on to learn about some of the practical tricks
- Feature scaling
  - If you have a problem with multiple features
  - You should make sure those features have a similar scale
    - Means gradient descent will converge more quickly
  - e.g.
    - $x_1$  = size (0 - 2000 feet)
    - $x_2$  = number of bedrooms (1-5)
    - Means the contours generated if we plot  $\theta_1$  vs.  $\theta_2$  give a very tall and thin shape due to the huge range difference
  - Running gradient descent on this kind of cost function can take a long time to find the global minimum



- Pathological input to gradient descent
  - So we need to rescale this input so it's more effective
  - So, if you define each value from  $x_1$  and  $x_2$  by dividing by the max for each feature
  - Contours become more like circles (as scaled between 0 and 1)
- May want to get everything into -1 to +1 range (approximately)
  - Want to avoid large ranges, small ranges or very different ranges from one another
  - Rule a thumb regarding acceptable ranges
    - -3 to +3 is generally fine - any bigger bad
    - -1/3 to +1/3 is ok - any smaller bad
- Can do **mean normalization**
  - Take a feature  $x_i$ 
    - Replace it by  $(x_i - \text{mean})/\text{max}$
    - So your values all have an average of about 0

$$x_i \leftarrow \frac{x_i - \mu_i}{s_i}$$

$\mu_i$  ← avg value of  $x_i$  in training set  
 $s_i$  ← range (max - min) (or standard deviation)

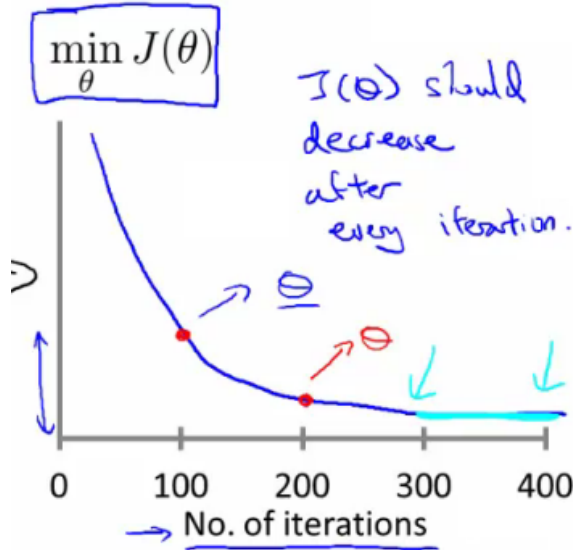
- Instead of max can also use standard deviation

## Learning Rate $\alpha$

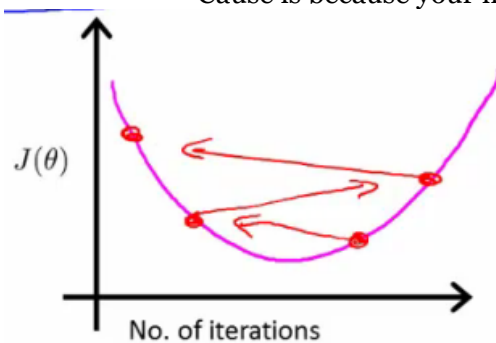
- Focus on the learning rate ( $\alpha$ )
- Topics
  - Update rule
  - Debugging
  - How to chose  $\alpha$

### **Make sure gradient descent is working**

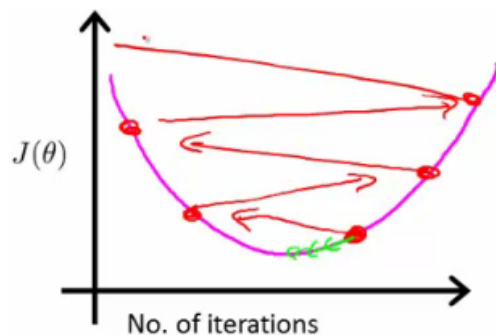
- Plot  $\min J(\theta)$  vs. no of iterations
  - (i.e. plotting  $J(\theta)$  over the course of gradient descent)
- If gradient descent is working then  $J(\theta)$  should decrease after every iteration
- Can also show if you're not making huge gains after a certain number
  - Can apply heuristics to reduce number of iterations if need be
  - If, for example, after 1000 iterations you reduce the parameters by nearly nothing you could chose to only run 1000 iterations in the future
  - Make sure you don't accidentally hard-code thresholds like this in and then forget about why they're their though!



- Number of iterations varies a lot
  - 30 iterations
  - 3000 iterations
  - 3000 000 iterations
  - Very hard to tel in advance how many iterations will be needed
  - Can often make a guess based a plot like this after the first 100 or so iterations
- Automatic convergence tests
  - Check if  $J(\theta)$  changes by a small threshold or less
    - Choosing this threshold is hard
    - So often easier to check for a straight line
      - Why? - Because we're seeing the straightness in the context of the whole algorithm
      - Could you design an automatic checker which calculates a threshold based on the systems preceding progress?
- Checking its working
  - If you plot  $J(\theta)$  vs iterations and see the value is increasing - means you probably need a smaller  $\alpha$ 
    - Cause is because your minimizing a function which looks like this



- But you overshoot, so reduce learning rate so you actually reach the minimum (green line)



- So, use a smaller  $\alpha$
- Another problem might be if  $J(\theta)$  looks like a series of waves
  - Here again, you need a smaller  $\alpha$
- However
  - If  $\alpha$  is small enough,  $J(\theta)$  will decrease on every iteration
  - BUT, if  $\alpha$  is too small then rate is too slow

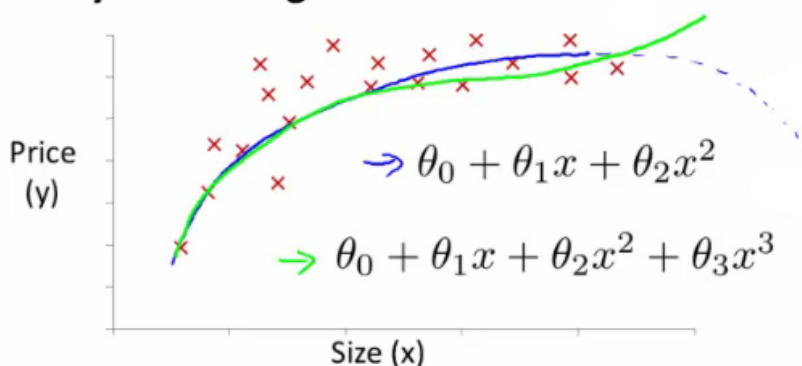
- A less steep incline is indicative of a slow convergence, because we're decreasing by less on each iteration than a steeper slope

- Typically
  - Try a range of alpha values
  - Plot  $J(\theta)$  vs number of iterations for each version of alpha
  - Go for roughly threefold increases
    - 0.001, 0.003, 0.01, 0.03, 0.1, 0.3

## Features and polynomial regression

- Choice of features and how you can get different learning algorithms by choosing appropriate features
- Polynomial regression for non-linear function
- Example
  - House price prediction
    - Two features
      - Frontage - width of the plot of land along road ( $x_1$ )
      - Depth - depth away from road ( $x_2$ )
  - You don't have to use just two features
    - **Can create new features**
  - Might decide that an important feature is the land area
    - So, create a new feature = frontage \* depth ( $x_3$ )
    - $h(x) = \theta_0 + \theta_1 x_3$ 
      - Area is a better indicator
  - Often, by defining new features you may get a better model
- Polynomial regression
  - May fit the data better
  - $\theta_0 + \theta_1 x + \theta_2 x^2$  e.g. here we have a quadratic function
  - For housing data could use a quadratic function
    - But may not fit the data so well - inflection point means housing prices decrease when size gets really big
    - So instead must use a cubic function

### Polynomial regression



- How do we fit the model to this data
  - To map our old linear hypothesis and cost functions to these polynomial descriptions the easy thing to do is set
    - $x_1 = x$
    - $x_2 = x^2$
    - $x_3 = x^3$
  - By selecting the features like this and applying the linear regression algorithms you can do polynomial linear regression
  - Remember, feature scaling becomes even more important here
- Instead of a conventional polynomial you could do variable  $^{1/\text{something}}$  - i.e. square root, cubed root etc
- Lots of features - later look at developing an algorithm to choose the best features

# Normal equation

- For some linear regression problems the normal equation provides a better solution
- So far we've been using gradient descent
  - Iterative algorithm which takes steps to converge
- Normal equation solves  $\theta$  analytically
  - Solve for the optimum value of theta
- Has some advantages and disadvantages

## How does it work?

- Simplified cost function
  - $J(\theta) = a\theta^2 + b\theta + c$ 
    - $\theta$  is just a real number, not a vector
  - Cost function is a quadratic function
  - How do you minimize this?
    - Do
      - $\frac{d}{d\theta} J(\theta) =$ 
        - Take derivative of  $J(\theta)$  with respect to  $\theta$
        - Set that derivative equal to 0
        - Allows you to solve for the value of  $\theta$  which minimizes  $J(\theta)$
- In our more complex problems;
  - Here  $\theta$  is an  $n+1$  dimensional vector of real numbers
  - Cost function is a function of the vector value
    - How do we minimize this function
      - Take the partial derivative of  $J(\theta)$  with respect  $\theta_j$  and set to 0 for every  $j$
      - Do that and solve for  $\theta_0$  to  $\theta_n$
      - This would give the values of  $\theta$  which minimize  $J(\theta)$
  - If you work through the calculus and the solution, the derivation is pretty complex
    - Not going to go through here
    - Instead, what do you need to know to implement this process

## Example of normal equation

Size (feet <sup>2</sup> )	Number of bedrooms	Number of floors	Age of home (years)	Price (\$1000)
$x_1$	$x_2$	$x_3$	$x_4$	$y$
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178

- Here
  - $m = 4$
  - $n = 4$
- To implement the normal equation
  - Take examples
  - Add an extra column ( $x_0$  feature)
  - Construct a matrix ( $X$  - **the design matrix**) which contains all the training data features in an  $[m \times n+1]$  matrix
  - Do something similar for  $y$ 
    - Construct a column vector  $y$  vector  $[m \times 1]$  matrix
  - Using the following equation ( $X$  transpose  $\times X$ ) inverse times  $X$  transpose  $y$ 

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

$$-1 \left( \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2104 & 1416 & 1534 & 852 \\ 5 & 3 & 3 & 2 \\ 1 & 2 & 2 & 1 \\ 45 & 40 & 30 & 36 \end{bmatrix} \times \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{bmatrix} \right) \times \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2104 & 1416 & 1534 & 852 \\ 5 & 3 & 3 & 2 \\ 1 & 2 & 2 & 1 \\ 45 & 40 & 30 & 36 \end{bmatrix} \times \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}$$

- If you compute this, you get the value of theta which minimize the cost function

### General case

- Have m training examples and n features
  - The **design matrix** (X)
    - Each training example is a n+1 dimensional feature column vector
    - X is constructed by taking each training example, determining its transpose (i.e. column -> row) and using it for a row in the design A
    - This creates an [m x (n+1)] matrix

$$\underline{x^{(i)}} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix} \in \mathbb{R}^{n+1} \quad \left| \quad \begin{matrix} \times \\ \text{(design} \\ \text{matrix)} \end{matrix} \right. = \begin{bmatrix} \text{---} (x^{(1)})^T \text{---} \\ \text{---} (x^{(2)})^T \text{---} \\ \vdots \\ \text{---} (x^{(m)})^T \text{---} \end{bmatrix}$$

- **Vector y**
  - Used by taking all the y values into a column vector

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

- What is this equation?!
  - $(X^T * X)^{-1}$ 
    - What is this --> the inverse of the matrix  $(X^T * X)$ 
      - i.e.  $A = X^T X$
      - $A^{-1} = (X^T X)^{-1}$
- In octave and MATLAB you could do;

`pinv(X'*x)*x'*y`

- X' is the notation for X transpose
- pinv is a function for the inverse of a matrix
- In a previous lecture discussed feature scaling
  - If you're using the normal equation then no need for feature scaling

### When should you use gradient descent and when should you use feature scaling?

- **Gradient descent**
  - Need to chose learning rate
  - Needs many iterations - could make it slower
  - Works well even when n is massive (millions)
    - Better suited to big data

- What is a big  $n$  though
    - 100 or even a 1000 is still (relativity) small
    - If  $n$  is 10 000 then look at using gradient descent
- **Normal equation**
  - No need to chose a learning rate
  - No need to iterate, check for convergence etc.
  - Normal equation needs to compute  $(X^T X)^{-1}$ 
    - This is the inverse of an  $n \times n$  matrix
    - With most implementations computing a matrix inverse grows by  $O(n^3)$ 
      - So not great
  - Slow of  $n$  is large
    - Can be much slower

## Normal equation and non-invertibility

- Advanced concept
  - Often asked about, but quite advanced, perhaps optional material
  - Phenomenon worth understanding, but not probably necessary
- When computing  $(X^T X)^{-1} * X^T * y$ 
  - What if  $(X^T X)$  is non-invertible (singular/degenerate)
    - Only some matrices are invertible
    - This should be quite a rare problem
      - Octave can invert matrices using
        - pinv (pseudo inverse)
          - This gets the right value even if  $(X^T X)$  is non-invertible
        - inv (inverse)
  - What does it mean for  $(X^T X)$  to be non-invertible
    - Normally two common causes
      - **Redundant features** in learning model
        - e.g.
          - $x_1$  = size in feet
          - $x_2$  = size in meters squared
      - **Too many features**
        - e.g.  $m \leq n$  ( $m$  is much larger than  $n$ )
          - $m = 10$
          - $n = 100$
        - Trying to fit 101 parameters from 10 training examples
        - Sometimes work, but not always a good idea
        - Not enough data
        - Later look at *why* this may be too little data
        - To solve this we
          - Delete features
          - Use **regularization** (let's you use lots of features for a small training set)
  - If you find  $(X^T X)$  to be non-invertible
    - Look at features --> are features linearly dependent?
      - So just delete one, will solve problem

# 06: Logistic Regression

[Previous](#) [Next](#) [Index](#)

## Classification

- Where  $y$  is a discrete value
  - Develop the logistic regression algorithm to determine what class a new input should fall into
- Classification problems
  - Email -> spam/not spam?
  - Online transactions -> fraudulent?
  - Tumor -> Malignant/benign
- Variable in these problems is  $Y$ 
  - $Y$  is either 0 or 1
    - 0 = negative class (absence of something)
    - 1 = positive class (presence of something)
- Start with **binary class problems**
  - Later look at multiclass classification problem, although this is just an extension of binary classification
- How do we develop a classification algorithm?
  - Tumour size vs malignancy (0 or 1)
  - We *could* use linear regression
    - Then threshold the classifier output (i.e. anything over some value is yes, else no)
    - In our example below linear regression with thresholding seems to work



- We can see above this does a reasonable job of stratifying the data points into one of two classes
  - But what if we had a single Yes with a very small tumour
  - This would lead to classifying all the existing yeses as nos
- Another issues with linear regression
  - We know  $Y$  is 0 or 1
  - Hypothesis can give values large than 1 or less than 0
- So, logistic regression generates a value where is always either 0 or 1
  - Logistic regression is a **classification algorithm** - don't be confused

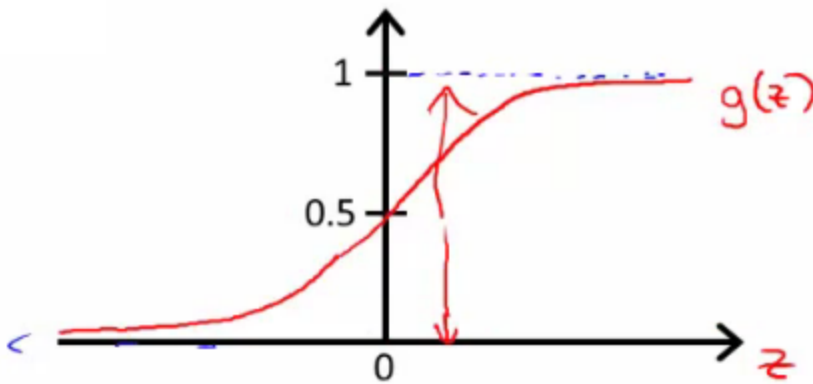


# Hypothesis representation

- What function is used to represent our hypothesis in classification
- We want our classifier to output values between 0 and 1
  - When using linear regression we did  $h_{\theta}(x) = (\theta^T x)$
  - For classification hypothesis representation we do  $h_{\theta}(x) = g(\theta^T x)$ 
    - Where we define  $g(z)$ 
      - $z$  is a real number
    - $g(z) = 1/(1 + e^{-z})$ 
      - This is the **sigmoid function**, or the **logistic function**
    - If we combine these equations we can write out the hypothesis as

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

- What does the sigmoid function look like
- Crosses 0.5 at the origin, then flattens out]
  - Asymptotes at 0 and 1



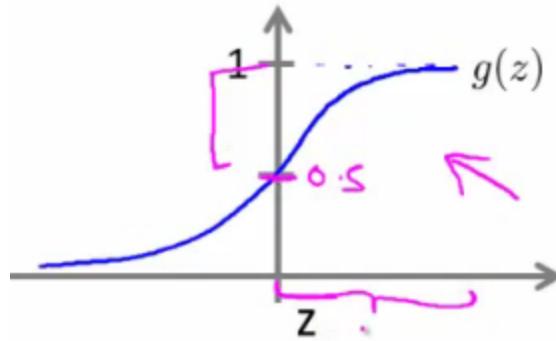
- Given this we need to fit  $\theta$  to our data

## Interpreting hypothesis output

- When our hypothesis ( $h_{\theta}(x)$ ) outputs a number, we treat that value as the estimated probability that  $y=1$  on input  $x$ 
  - Example
    - If  $X$  is a feature vector with  $x_0 = 1$  (as always) and  $x_1 = \text{tumourSize}$
    - $h_{\theta}(x) = 0.7$ 
      - Tells a patient they have a 70% chance of a tumor being malignant
  - We can write this using the following notation
    - $h_{\theta}(x) = P(y=1|x; \theta)$
  - What does this mean?
    - Probability that  $y=1$ , given  $x$ , parameterized by  $\theta$
- Since this is a binary classification task we know  $y = 0$  or  $1$ 
  - So the following must be true
    - $P(y=1|x; \theta) + P(y=0|x; \theta) = 1$
    - $P(y=0|x; \theta) = 1 - P(y=1|x; \theta)$

# Decision boundary

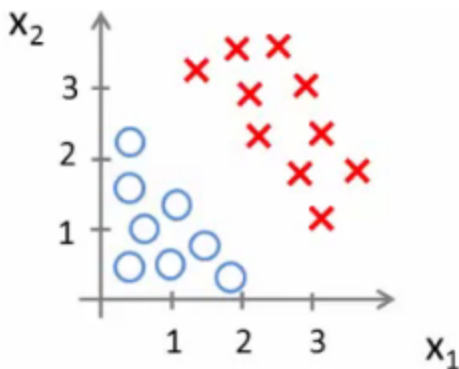
- Gives a better sense of what the hypothesis function is computing
- Better understand of what the hypothesis function looks like
  - One way of using the sigmoid function is;
    - When the probability of  $y$  being 1 is greater than 0.5 then we can predict  $y = 1$
    - Else we predict  $y = 0$
  - When is it exactly that  $h_{\theta}(x)$  is greater than 0.5?
    - Look at sigmoid function
      - $g(z)$  is greater than or equal to 0.5 when  $z$  is greater than or equal to 0



- So if  $z$  is positive,  $g(z)$  is greater than 0.5
    - $z = (\theta^T x)$
  - So when
    - $\theta^T x \geq 0$
  - Then  $h_{\theta} \geq 0.5$
- So what we've shown is that the hypothesis predicts  $y = 1$  when  $\theta^T x \geq 0$ 
  - The corollary of that when  $\theta^T x \leq 0$  then the hypothesis predicts  $y = 0$
  - Let's use this to better understand how the hypothesis makes its predictions

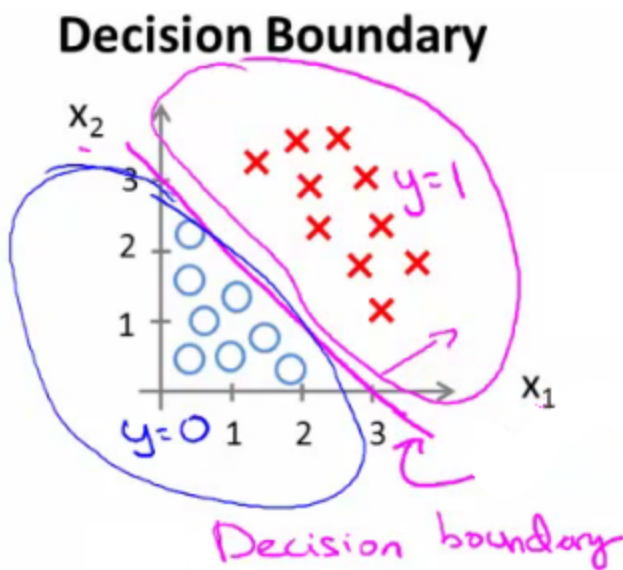
## Decision boundary

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



- So, for example
  - $\theta_0 = -3$

- $\theta_1 = 1$
  - $\theta_2 = 1$
- So our parameter vector is a column vector with the above values
  - So,  $\theta^T$  is a row vector =  $[-3, 1, 1]$
- What does this mean?
  - The  $z$  here becomes  $\theta^T x$
  - We predict " $y = 1$ " if
    - $-3x_0 + 1x_1 + 1x_2 \geq 0$
    - $-3 + x_1 + x_2 \geq 0$
- We can also re-write this as
  - If  $(x_1 + x_2 \geq 3)$  then we predict  $y = 1$
  - If we plot
    - $x_1 + x_2 = 3$  we graphically plot our **decision boundary**

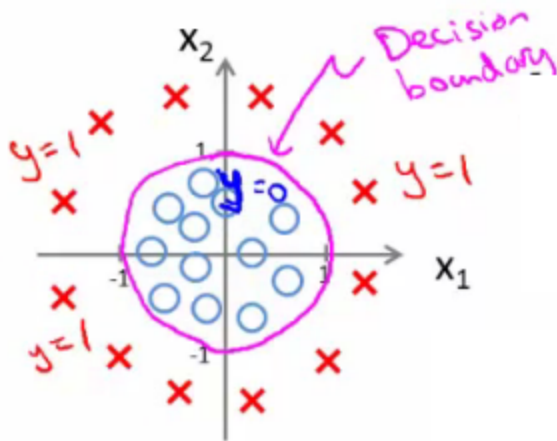


- Means we have these two regions on the graph
  - Blue = false
  - Magenta = true
  - Line = decision boundary
    - Concretely, the straight line is the set of points where  $h_\theta(x) = 0.5$  exactly
  - The decision boundary is a property of the hypothesis
    - Means we can create the boundary with the hypothesis and parameters without any data
      - Later, we use the data to determine the parameter values
    - i.e.  $y = 1$  if
      - $5 - x_1 > 0$
      - $5 > x_1$

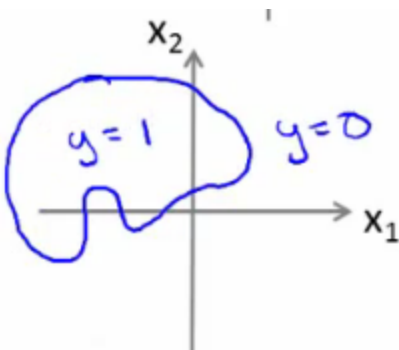
## Non-linear decision boundaries

- Get logistic regression to fit a complex non-linear data set
  - Like polynomial regress add higher order terms
  - So say we have

- $h_{\theta}(x) = g(\theta_0 + \theta_1 x_1 + \theta_3 x_1^2 + \theta_4 x_2^2)$
- We take the transpose of the  $\theta$  vector times the input vector
  - Say  $\theta^T$  was  $[-1, 0, 0, 1, 1]$  then we say;
  - Predict that "y = 1" if
    - $-1 + x_1^2 + x_2^2 \geq 0$
    - or
    - $x_1^2 + x_2^2 \geq 1$
  - If we plot  $x_1^2 + x_2^2 = 1$ 
    - This gives us a circle with a radius of 1 around 0



- Mean we can build more complex decision boundaries by fitting complex parameters to this (relatively) simple hypothesis
- More complex decision boundaries?
  - By using higher order polynomial terms, we can get even more complex decision boundaries



## Cost function for logistic regression

- Fit  $\theta$  parameters
- Define the optimization object for the cost function we use the fit the parameters
  - Training set of  $m$  training examples
    - Each example has is  $n+1$  length column vector

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

m examples  $x \in \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_n \end{bmatrix} \quad x_0 = 1, y \in \{0, 1\}$

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

- This is the situation
  - Set of m training examples
  - Each example is a feature vector which is n+1 dimensional
  - $x_0 = 1$
  - $y \in \{0, 1\}$
  - Hypothesis is based on parameters ( $\theta$ )
    - Given the training set how to we chose/fit  $\theta$ ?
- Linear regression uses the following function to determine  $\theta$

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

- Instead of writing the squared error term, we can write
  - If we define "cost()" as;
    - $\text{cost}(h_{\theta}(x^i), y) = 1/2(h_{\theta}(x^i) - y^i)^2$
    - Which evaluates to the cost for an individual example using the same measure as used in linear regression
  - We can **redefine  $J(\theta)$**  as

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

- Which, appropriately, is the sum of all the individual costs over the training data (i.e. the same as linear regression)
- To further simplify it we can get rid of the superscripts
  - So

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

- What does this actually mean?

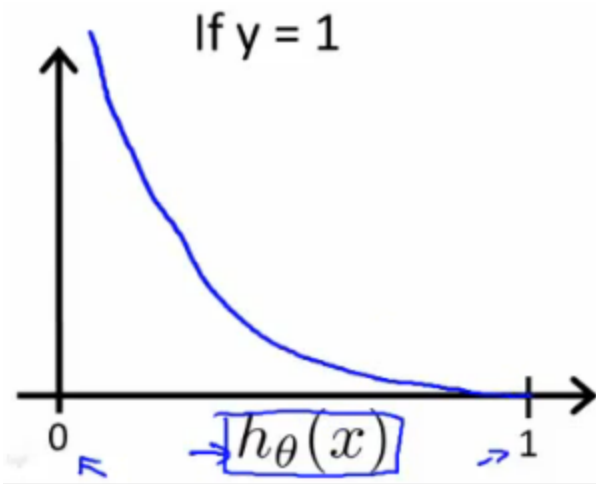
- This is the cost you want the learning algorithm to pay if the outcome is  $h_\theta(x)$  and the actual outcome is  $y$
  - If we use this function for logistic regression this is a **non-convex function** for parameter optimization
    - Could work....
- What do we mean by non convex?
  - We have some function -  $J(\theta)$  - for determining the parameters
  - Our hypothesis function has a non-linearity (sigmoid function of  $h_\theta(x)$ )
    - This is a complicated non-linear function
  - If you take  $h_\theta(x)$  and plug it into the Cost() function, and then plug the Cost() function into  $J(\theta)$  and plot  $J(\theta)$  we find many local optimum -> *non convex function*
  - Why is this a problem
    - Lots of local minima mean gradient descent may not find the global optimum - may get stuck in a local minimum
  - We would like a convex function so if you run gradient descent you converge to a global minimum

### A convex logistic regression cost function

- To get around this we need a different, convex Cost() function which means we can apply gradient descent

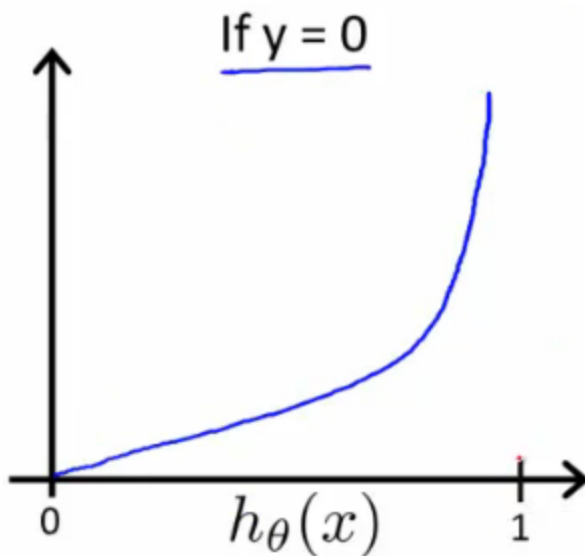
$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

- **This is our logistic regression cost function**
  - This is the penalty the algorithm pays
  - Plot the function
- Plot  $y = 1$ 
  - So  $h_\theta(x)$  evaluates as  $-\log(h_\theta(x))$



- So when we're right, cost function is 0

- Else it slowly increases cost function as we become "more" wrong
  - X axis is what we predict
  - Y axis is the cost associated with that prediction
- This cost functions has some interesting properties
  - If  $y = 1$  and  $h_{\theta}(x) = 1$ 
    - If hypothesis predicts exactly 1 and thats exactly correct then that corresponds to 0 (exactly, not nearly 0)
  - As  $h_{\theta}(x)$  goes to 0
    - Cost goes to infinity
    - This captures the intuition that if  $h_{\theta}(x) = 0$  (predict  $P(y=1|x; \theta) = 0$ ) but  $y = 1$  this will penalize the learning algorithm with a massive cost
- What about if  $y = 0$
- then cost is evaluated as  $-\log(1 - h_{\theta}(x))$ 
  - Just get inverse of the other function



- Now it goes to plus infinity as  $h_{\theta}(x)$  goes to 1
- With our particular cost functions  $J(\theta)$  is going to be convex and avoid local minimum

## Simplified cost function and gradient descent

- Define a simpler way to write the cost function and apply gradient descent to the logistic regression
  - By the end should be able to implement a fully functional logistic regression function
- Logistic regression cost function is as follows

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

$$\text{Cost}(h_{\theta}(x), y) = \begin{cases} -\log(h_{\theta}(x)) & \text{if } y = 1 \\ -\log(1 - h_{\theta}(x)) & \text{if } y = 0 \end{cases}$$

Note:  $y = 0$  or  $1$  always

- This is the cost for a single example
  - For binary classification problems  $y$  is always 0 or 1
    - Because of this, we can have a simpler way to write the cost function
      - Rather than writing cost function on two lines/two cases
      - Can compress them into one equation - more efficient
  - Can write cost function is
    - **$\text{cost}(\mathbf{h}_{\theta}, (\mathbf{x}), y) = -y \log(\mathbf{h}_{\theta}(\mathbf{x})) - (1-y) \log(1 - \mathbf{h}_{\theta}(\mathbf{x}))$** 
      - This equation is a more compact of the two cases above
  - We know that there are only two possible cases
    - $y = 1$ 
      - Then our equation simplifies to
        - $-\log(h_{\theta}(x)) - (0)\log(1 - h_{\theta}(x))$ 
          - $-\log(h_{\theta}(x))$
          - Which is what we had before when  $y = 1$
      - $y = 0$ 
        - Then our equation simplifies to
          - $-(0)\log(h_{\theta}(x)) - (1)\log(1 - h_{\theta}(x))$
          - $= -\log(1 - h_{\theta}(x))$
          - Which is what we had before when  $y = 0$
      - Clever!
  - So, in summary, our cost function for the  $\theta$  parameters can be defined as

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

- Why do we chose this function when other cost functions exist?
  - This cost function can be derived from statistics using the principle of **maximum likelihood estimation**
    - Note this does mean there's an underlying Gaussian assumption relating to the distribution of features
  - Also has the nice property that it's convex
- To fit parameters  $\theta$ :
  - Find parameters  $\theta$  which minimize  $J(\theta)$
  - This means we have a set of parameters to use in our model for future predictions
- Then, if we're given some new example with set of features  $x$ , we can take the  $\theta$  which we generated, and output our prediction using



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

- This result is
  - $p(y=1 | x; \theta)$ 
    - Probability  $y = 1$ , given  $x$ , parameterized by  $\theta$

## How to minimize the logistic regression cost function

- Now we need to figure out how to minimize  $J(\theta)$ 
  - Use gradient descent as before
  - Repeatedly update each parameter using a learning rate

Repeat {

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

} (simultaneously update all  $\theta_j$ )

- If you had  $n$  features, you would have an  $n+1$  column vector for  $\theta$
- This equation is the same as the linear regression rule
  - The only difference is that our definition for the hypothesis has changed
- Previously, we spoke about how to monitor gradient descent to check it's working
  - Can do the same thing here for logistic regression
- When implementing logistic regression with gradient descent, we have to update all the  $\theta$  values ( $\theta_0$  to  $\theta_n$ ) simultaneously
  - Could use a for loop
  - Better would be a vectorized implementation
- Feature scaling for gradient descent for logistic regression also applies here

## Advanced optimization

- Previously we looked at gradient descent for minimizing the cost function
- Here look at advanced concepts for minimizing the cost function for logistic regression
  - Good for large machine learning problems (e.g. huge feature set)
- *What is gradient descent actually doing?*
  - We have some cost function  $J(\theta)$ , and we want to minimize it
  - We need to write code which can take  $\theta$  as input and compute the following
    - $J(\theta)$
    - Partial derivative of  $J(\theta)$  with respect to  $j$  (where  $j=0$  to  $j = n$ )

$$J(\theta)$$

$$\frac{\partial}{\partial \theta_j} J(\theta) \quad (\text{for } j = 0, 1, \dots, n)$$

- Given code that can do these two things
  - Gradient descent repeatedly does the following update

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

- So update each  $j$  in  $\theta$  sequentially
- So, we must;
  - Supply code to compute  $J(\theta)$  and the derivatives
  - Then plug these values into gradient descent
- Alternatively, instead of gradient descent to minimize the cost function we could use
  - **Conjugate gradient**
  - **BFGS** (Broyden-Fletcher-Goldfarb-Shanno)
  - **L-BFGS** (Limited memory - BFGS)
- These are more optimized algorithms which take that same input and minimize the cost function
- These are *very* complicated algorithms
- Some properties
  - **Advantages**
    - No need to manually pick alpha (learning rate)
      - Have a clever inner loop (line search algorithm) which tries a bunch of alpha values and picks a good one
    - Often faster than gradient descent
      - Do more than just pick a good learning rate
    - Can be used successfully without understanding their complexity
  - **Disadvantages**
    - Could make debugging more difficult
    - Should not be implemented themselves
    - Different libraries may use different implementations - may hit performance

## Using advanced cost minimization algorithms

- How to use algorithms
  - Say we have the following example

Example:

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$$

$$J(\theta) = (\theta_1 - 5)^2 + (\theta_2 - 5)^2$$

$$\frac{\partial}{\partial \theta_1} J(\theta) = 2(\theta_1 - 5)$$

$$\frac{\partial}{\partial \theta_2} J(\theta) = 2(\theta_2 - 5)$$

- Example above
  - $\theta_1$  and  $\theta_2$  (two parameters)

- Cost function here is  $J(\theta) = (\theta_1 - 5)^2 + (\theta_2 - 5)^2$
- The derivatives of the  $J(\theta)$  with respect to either  $\theta_1$  and  $\theta_2$  turns out to be the  $2(\theta_i - 5)$
- First we need to define our cost function, which should have the following signature

```
function [jval, gradient] = costFunction(THETA)
```

- Input for the cost function is **THETA**, which is a vector of the  $\theta$  parameters
- Two return values from **costFunction** are
  - **jval**
    - How we compute the cost function  $\theta$  (the underived cost function)
      - In this case  $= (\theta_1 - 5)^2 + (\theta_2 - 5)^2$
  - **gradient**
    - 2 by 1 vector
    - 2 elements are the two partial derivative terms
    - i.e. this is an n-dimensional vector
      - Each indexed value gives the partial derivatives for the partial derivative of  $J(\theta)$  with respect to  $\theta_i$
      - Where  $i$  is the index position in the **gradient** vector
- With the cost function implemented, we can call the advanced algorithm using

```
options= optimset('GradObj', 'on', 'MaxIter', '100'); % define the
options data structure
initialTheta= zeros(2,1); # set the initial dimensions for theta %
initialize the theta values
[optTheta, funtionVal, exitFlag]= fminunc(@costFunction,
initialTheta, options); % run the algorithm
```

- Here
  - **options** is a data structure giving options for the algorithm
  - **fminunc**
    - function minimize the cost function (find **minimum** of **unconstrained** multivariable function)
  - **@costFunction** is a pointer to the costFunction function to be used
- For the octave implementation
  - **initialTheta** must be a matrix of at least two dimensions
- How do we apply this to logistic regression?
  - Here we have a vector

$$\text{theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$$

```
function [jVal, gradient] = costFunction(theta)
```

```
    jVal = [code to compute  $J(\theta)$ ];
```

```
    gradient(1) = [code to compute  $\frac{\partial}{\partial \theta_0} J(\theta)$ ];
```

```
    gradient(2) = [code to compute  $\frac{\partial}{\partial \theta_1} J(\theta)$ ];
```

```
    ⋮
```

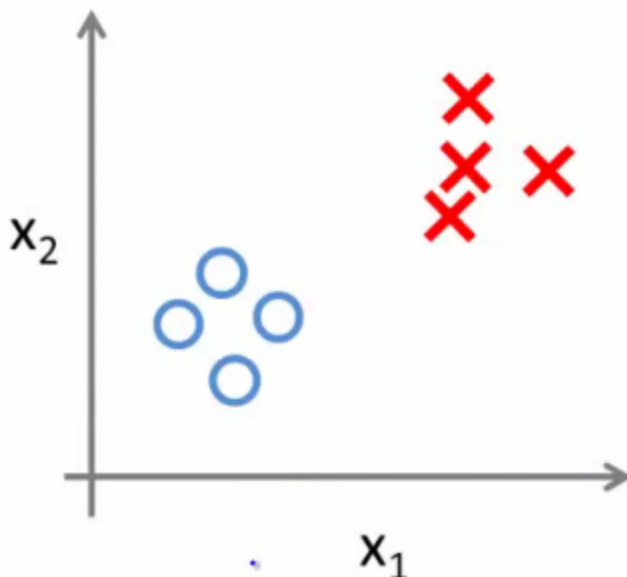
```
    gradient(n+1) = [code to compute  $\frac{\partial}{\partial \theta_n} J(\theta)$ ];
```

- Here
  - theta is a n+1 dimensional column vector
  - Octave indexes from 1, not 0
- Write a cost function which captures the cost function for logistic regression

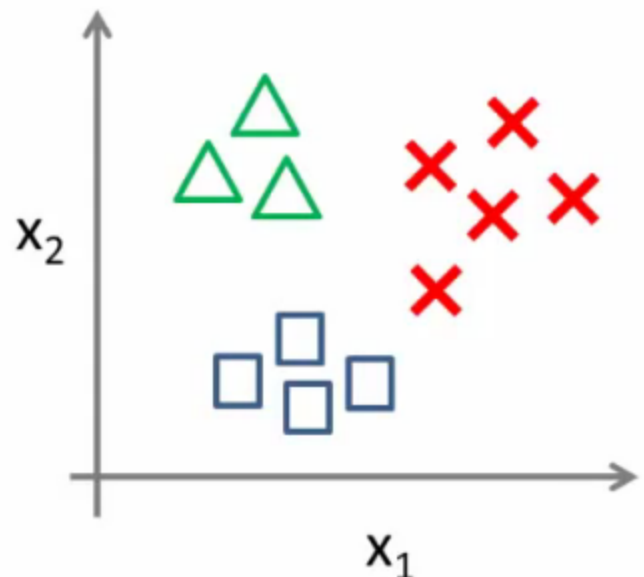
## Multiclass classification problems

- Getting logistic regression for multiclass classification using **one vs. all**
- Multiclass - more than yes or no (1 or 0)
  - Classification with multiple classes for assignment

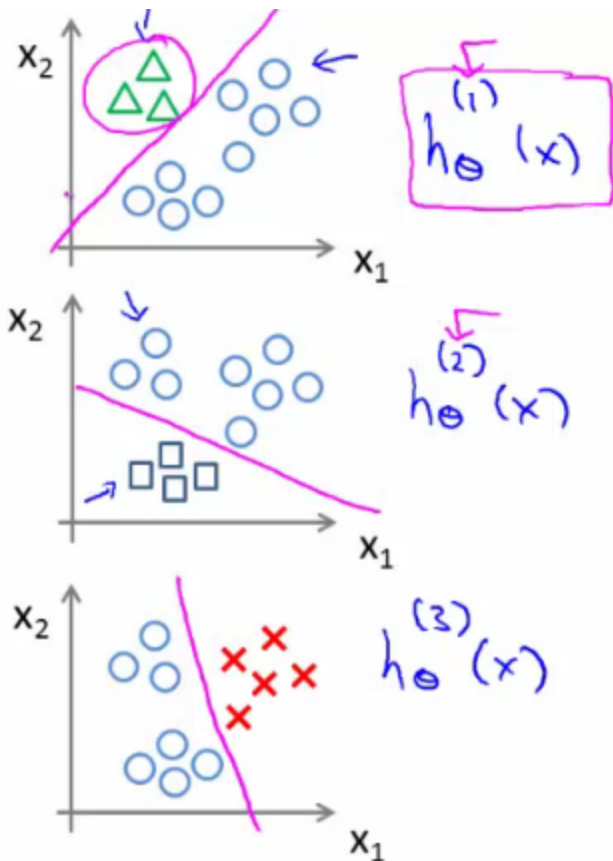
Binary classification:



Multi-class classification:



- Given a dataset with three classes, how do we get a learning algorithm to work?
  - Use one vs. all classification make binary classification work for multiclass classification
- One vs. all classification**
  - Split the training set into three separate binary classification problems
    - i.e. create a new fake training set
      - Triangle (1) vs crosses and squares (0)  $h_{\theta}^1(x)$ 
        - $P(y=1 \mid x_1; \theta)$
      - Crosses (1) vs triangle and square (0)  $h_{\theta}^2(x)$ 
        - $P(y=1 \mid x_2; \theta)$
      - Square (1) vs crosses and square (0)  $h_{\theta}^3(x)$ 
        - $P(y=1 \mid x_3; \theta)$



- Overall**
  - Train a logistic regression classifier  $h_{\theta}^{(i)}(x)$  for each class  $i$  to predict the probability that  $y = i$
  - On a new input,  $x$  to make a prediction, pick the class  $i$  that maximizes the probability that  $h_{\theta}^{(i)}(x) = 1$