Andrew Ng's Machine Learning Course

Week 1

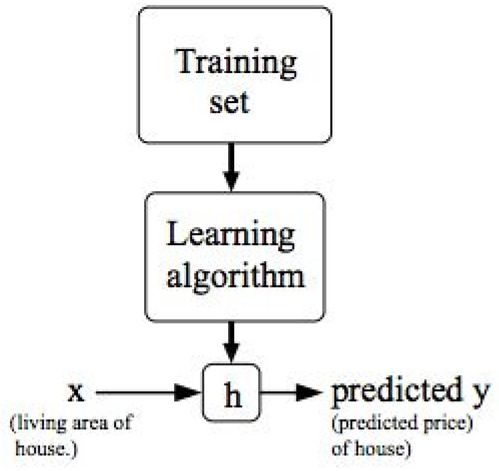
Introduction

- 2 Machine Learning definitions
  - Informal (by Samuel, 1950): field of study that gives computers the ability to learn without being explicitly programmed
  - Formal: A program is said to learn from experience E with respect to task T and measure P if its performance on T, as measured by P, improves with experience E
- There are 2 major machine learning algorithm categories:
  - *Supervised Learning:* we are given data that already has the correct answer (i.e whether a tumor is malignant or benign or the price at what a house sold). Problems here are classified in:
    - Regression: we are trying to predict results within a continuous output, we are mapping input variables to a continuous function (i.e. given the size of the house, predict the price the price the house sold for. Price as a function of size is a continuous function). Regression is about predicting a quantity
    - Classification: we are trying to predict results with a discrete output. We are mapping input variables to discrete categories (i.e. whether the tumor is malignant or benign). Classification is about predicting a label
  - *Unsupervised Learning*: we are given data and then asked: can you find structure in this data? Problems can be classified in
    - Clustering: algorithm groups the data in different clusters. An example is google news grouping stories from different sources into one topic (i.e. many stories under a 'BP Oil spill' topic). Other examples: genomics, data center machines clustering, market segmentation
    - Non-clustering: the cocktail party algorithm helps us to find structure in a chaotic environment. An example is a cocktail party where many people are talking and the algorithm identifies individual voices and people from a mesh of sounds

Linear Regression with one variable (univariate linear regression)

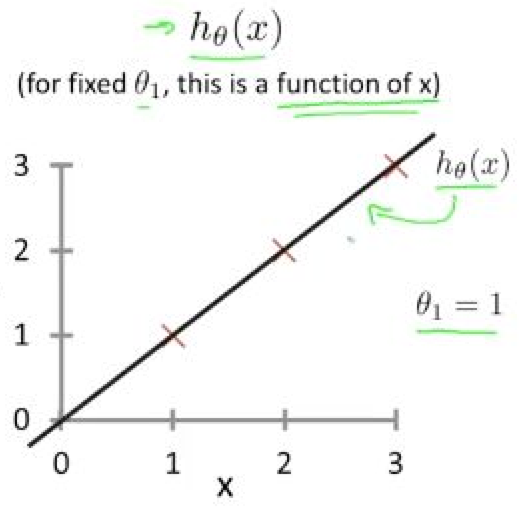
- Regression predicts a real-valued output, i.e. the price of a house

- Model Representation
  - Dataset given is called the training set.
  - A training set has
    - a number of rows (m)
    - x which is the input variable
    - y which is the output variable (the variable I'm trying to predict)
    - i is the index that denotes one row in the dataset
    - (x(i),y(i)) denotes row i in my training set
  - We are going to predict that y is a linear function of x
  - Given this, a supervised learning problem can be defined as having a goal of, given a training set, to learn a function h (h stands for hypothesis): X → Y so that h(x) is a good predictor for the corresponding value of y. This can be expressed as:



  - H can also be expressed as the following linear function: h_θ (x) = $\theta_0 + \theta_1 x$
    - Parameters: θ_0 and θ_1
  - When the target variable y is a continuous value we call it a regression problem. If the target variable only takes a small number of values (i.e. benign or malignant) we call it classification
- θ Cost function
  - Idea is to to choose parameters so that h(x) is close to y for all our (x,y) examples
  - We are trying to measure the accuracy of our hypothesis function using a cost function
  - In general, we are trying to minimize θ_0 and θ_1 because I want the difference between $h(x)$ (*predicted value*) and y (*actual value*) to be small
  - This function is called the cost function and is expressed as J(θ_0, θ_1) $= \frac{1}{2m} \sum_{i=1}^{m} ((h(x^{(i)} - y^{(i)}))^2$
  - This function is also called the squared error function and is the most commonly used function for regression problems

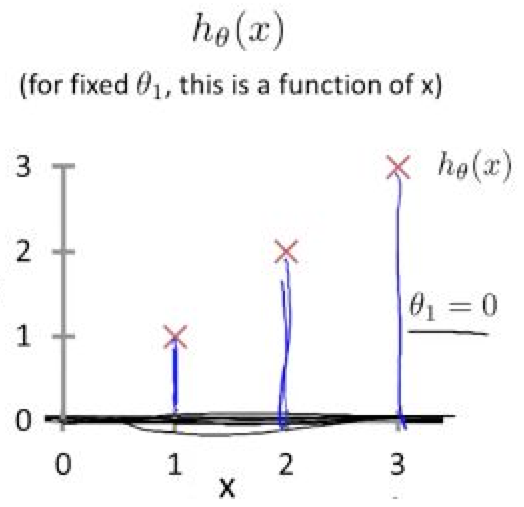
- Intuition I
  - We make the cost function only have one variable: θ_1 which then makes my hypothesis equation as h_θ (x) = $\theta_1 x$
  - If we make θ_1 fixed then h_θ (x) is a function of x
  - For example if θ_1 = 1 then h_θ (x) is just the x value, which we can plot as shown below



Why? Because h_θ (x) is equal to the x value so if x=1 then y=1, if x=2 then y=2 and so on. For this training set this means the slope goes right through them and the difference (or cost function) between $h(x)$ (predicted value) and y (actual value) is pretty much 0 because our function when θ_1 = 1 can be written as: J(θ_1) $= \frac{1}{2m}\sum_{i=1}^{m}((h(x^{(i)} - y^{(i)}))^2$ =

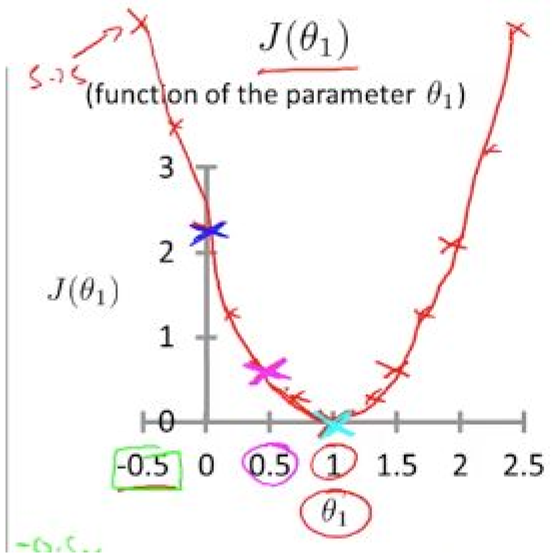
$\frac{1}{2m}\sum_{i=1}^{3}$ $(1-1)^2$ + $(2-2)^2$ + $(3-3)^2$ = 0 + 0 + 0 = 0 so J(θ_1= 1) = 0

  - If θ_1= 0 then our hypothesis function is plotted as:



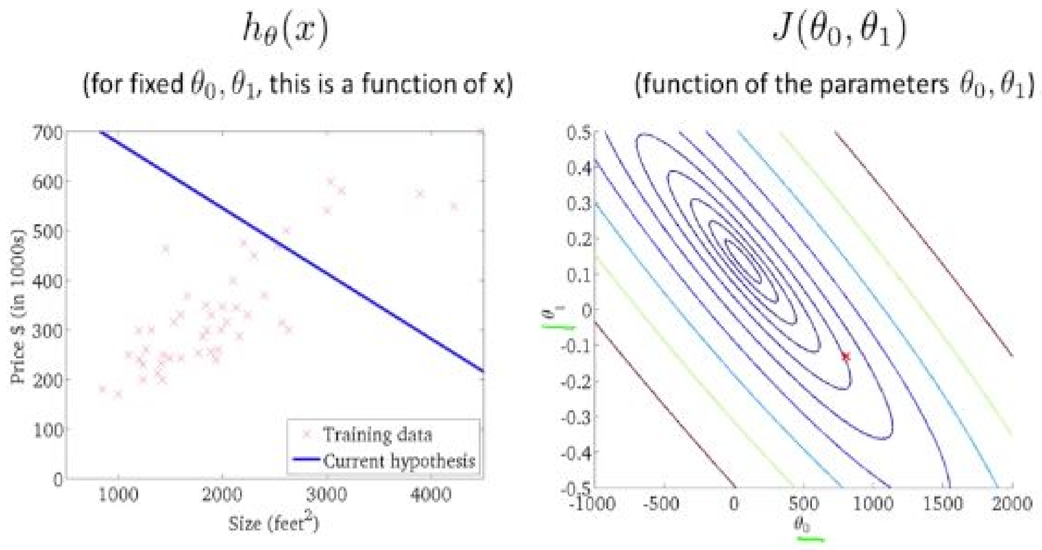
In this case the slope is 0 or, remembering our hypothesis function we have h_θ (x) = $\theta_1 x$ = h_θ (x) = $0 * x$ = 0. Our slope in this case is then the x axis and our cost function J (θ_1) = $\frac{1}{2m}\sum_{i=1}^{m}((h(x^{(i)} - y^{(i)}))^2$ = $\frac{1}{2(3)}\sum_{i=1}^{3}$ $(0-1)^2$ + $(0-2)^2$ + $(0-3)^2$ = $\frac{1}{6}$ (1+4+9) = 14/6

- We can continue to get more values for the J function and we would end up with a graph such as this:



- Our optimization is to find the J value at where the difference between $h_\theta(x)$ and x is minimal, which in this case happens to be $\theta_1 = 1$
- Another way to look at this is that we have a training set of scattered points and we are trying to find a line (defined by $h_\theta(x)$) that passes through these points

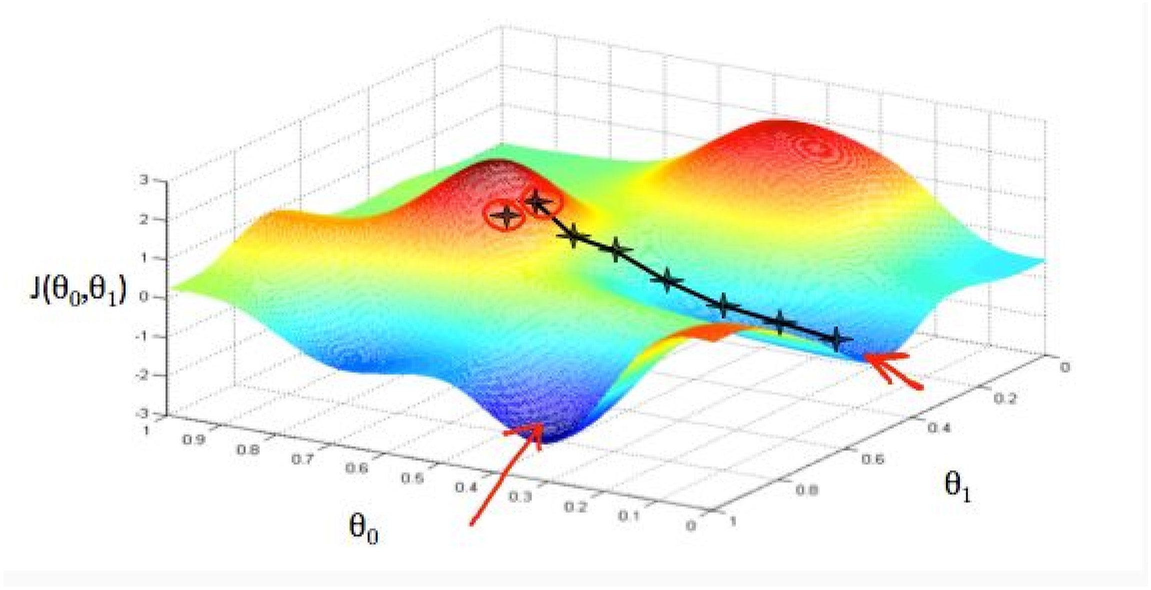
○ Intuition II
  - In this case we don't equal θ_0 = 0 and the resulting J function is still a parabola
  - Use contour plots to graph a 3 dimensional visualization of the cost function J(θ_0, θ_1)
  - In the contour plot we see concentric circles that deviate from the minimum cost.
  - Each concentric circle represents a value of J(θ_0, θ_1)
  - An example of a concentric value showing a value of J and the corresponding hypothesis is



- This particular example shows a point that is far from the minimum and therefore is a bad fit for the training set example on the left plot (we can see that because $h(x)$ does not pass close to many of the points in the training set)

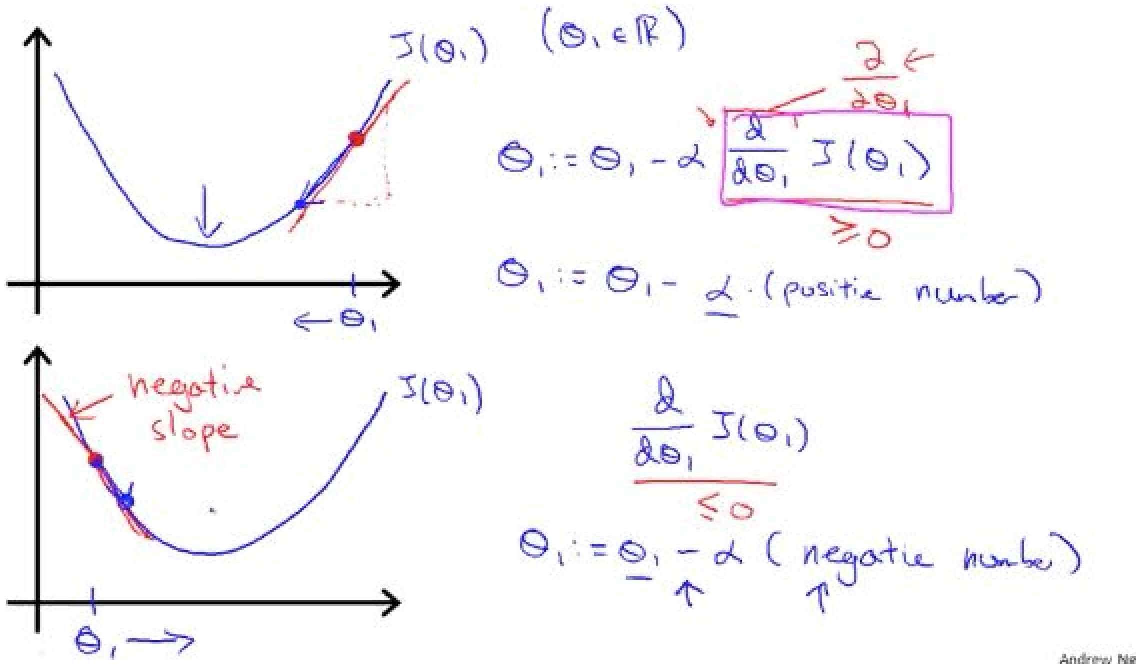
Linear Regression with one variable (Gradient Descent)

- Gradient Descent is an algorithm used to find the lowest cost function by starting at a point (θ_0, θ_1) and then 'going downhill' until we get at the bottom of the pit. An example is the following (the path shown is gradient descent at work)



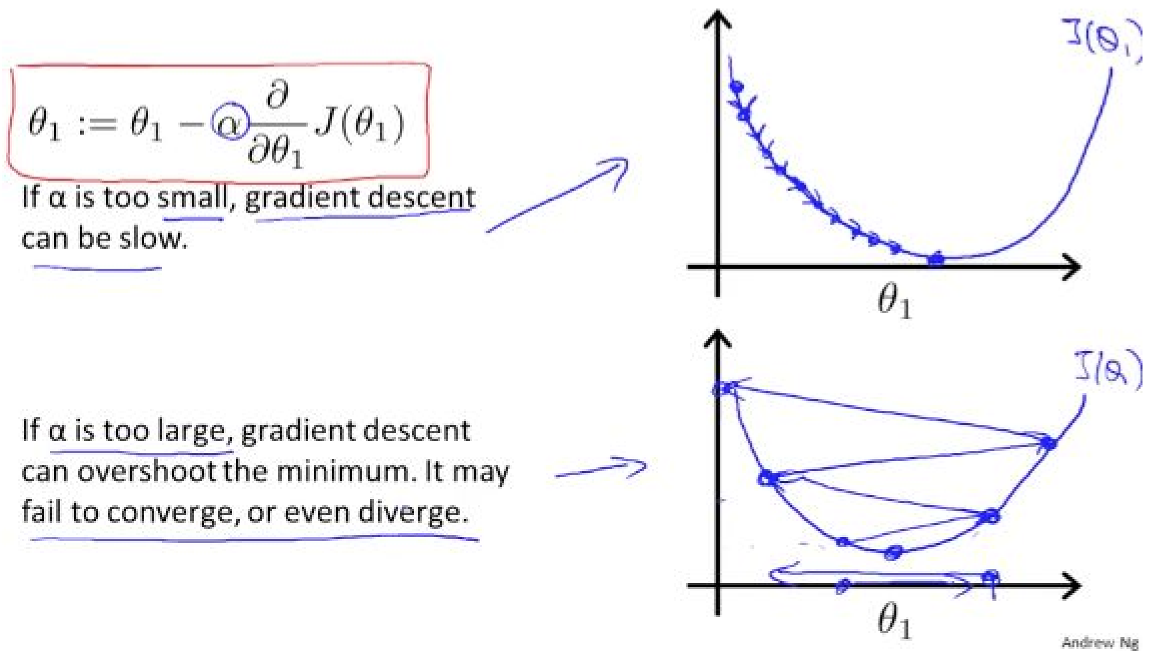
- The algorithm is then: repeat until convergence $\{ \theta_0 := \theta_j - \alpha \frac{\delta}{\delta\theta_j} J(\theta_0, \theta_1) \}$ where
  - j represents the iteration number
  - α is the learning rate (or how big a step we take on each iteration) and

    - $\frac{\delta}{\delta\theta_j} J(\theta_0, \theta_1)$ is the derivative term
- At each iteration we simultaneously update the parameters θ_0 and θ_1, updating one parameter and then doing another iteration would yield the wrong implementation. This is called simultaneous update and it goes with gradient descent
- Correct way:
    - temp0 := $\theta_0 - \alpha\frac{\delta}{\delta\theta_0} J(\theta_0, \theta_1)$
    - temp1 = $\theta_1 - \alpha\frac{\delta}{\delta\theta_1} J(\theta_0, \theta_1)$
    - θ_0 := temp0
    - θ_1 := temp1
- Incorrect way:
    - temp0 := $\theta_0 - \alpha\frac{\delta}{\delta\theta_0} J(\theta_0, \theta_1)$
    - θ_0 := temp0
    - temp1 = $\theta_1 - \alpha\frac{\delta}{\delta\theta_1} J(\theta_0, \theta_1)$
    - θ_1 := temp1
- Gradient Descent Intuition
    - is the idea that our equation $\{ \theta_0 := \theta_j - \alpha\frac{\delta}{\delta\theta_j} J(\theta_0, \theta_1)$ } will converge to the minimum regardless of where you start. If you start at the left side of the minimum the algorithm will suggest a step to the right. If you start at the right side the algorithm will suggest an iteration to the left
    - Why? Because the derivative term is the slope of the tangent line that touches the current iteration point. In the first chart below we see that the derivative is going to be a positive number and therefore $\alpha * \textit{positive number}$ will generate a positive number which is then subtracted from the current iteration effectively creating a new iteration to the left.



Conversely, in the 2nd chart we start on the left side of the function and the derivate (or slope) will be negative. Multiplying $\alpha * \textit{negative number}$ gives us a negative number, which then is subtracted from the current iteration. Subtracting a negative number means adding it and therefore the new iteration is positive, moving our θ to the right. Moving it to the right is is the right thing since I want to be at the minimum point in the function

- Learning rate α: if α is too small, gradient descent will be slow. If α is too big we might overshoot and fail to converge or even diverge as shown below

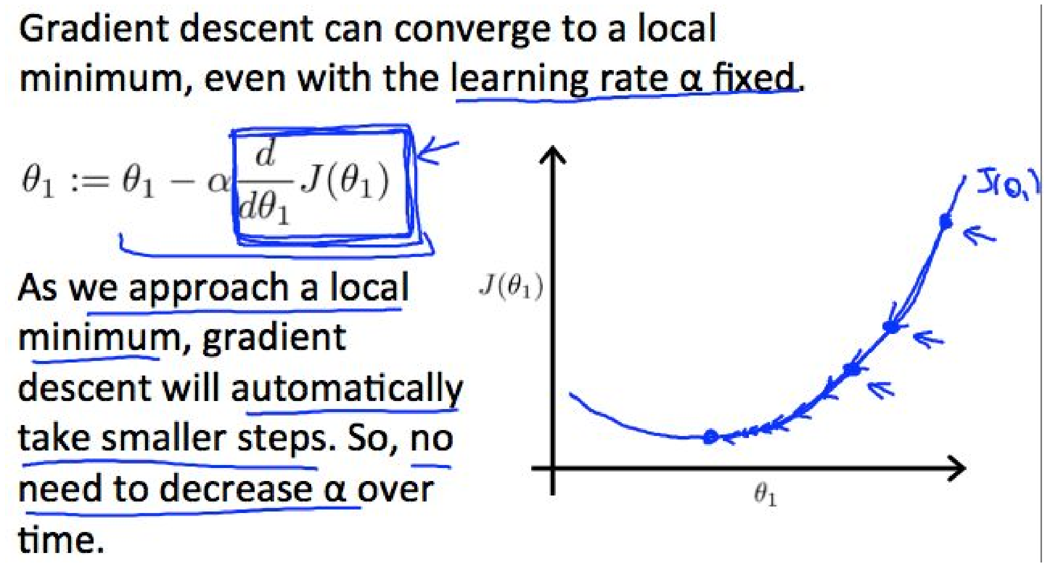


$$\boxed{\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)}$$

If α is too small, gradient descent can be slow.

If α is too large, gradient descent can overshoot the minimum. It may fail to converge, or even diverge.

   - What does gradient descent do if I am already at a minimum? Nothing because in that case the derivative (or slope) is = 0, therefore the equation here becomes $\theta_1 := \theta_1 - \alpha * 0 = \theta_1 := \theta_1 - 0\ \theta_1 := \theta_1$
   - Gradient descent will converge even if the learning rate α is fixed because the slope or derivative from one iteration to another becomes smaller:



Gradient descent can converge to a local minimum, even with the learning rate α fixed.

$$\theta_1 := \theta_1 - \alpha \boxed{\frac{d}{d\theta_1} J(\theta_1)}$$

As we approach a local minimum, gradient descent will automatically take smaller steps. So, no need to decrease α over time.

- Gradient Descent for Linear Regression
   - When we apply the gradient descent algorithm to linear regression we have a new equation: repeat until convergence $\{\ \theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_0(x_i) - y_i),\ \theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_0(x_i) - y_i\ ) x_i)\ \}$
   - Applying these equations to our initial hypothesis will make the hypothesis more accurate
   - This is called 'batch' gradient descent because in each iteration we are looking at the whole training set (aka a batch of points)

Linear Algebra Review

- Matrix is a rectangular array of numbers like this one

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix}$$

- Dimension of a matrix is the number of rows * the number of columns
- Element A_{ij} of a matrix is the element in row i, column j
- A vector is a special case of a matrix: an n x 1 matrix like this one:

$$\begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix}$$

- We will be using 1-indexed vectors
- Matrix Addition:
  - To add 2 matrices you add up the corresponding elements for each matrix as shown here:

    $$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} a+w & b+x \\ c+y & d+z \end{bmatrix}$$

  - The ***matrices have to be of the same dimension***. For example, you can add 2 matrices that are of dimension 3 x 2 but you cannot add a matrix of 3 x 2 with a matrix of 2 x 3
- Scalar Multiplication
  - Multiplying a matrix by a real number, the number could be an int or or a fraction as shown here:

    $$\begin{bmatrix} a & b \\ c & d \end{bmatrix} * x = \begin{bmatrix} a*x & b*x \\ c*x & d*x \end{bmatrix}$$

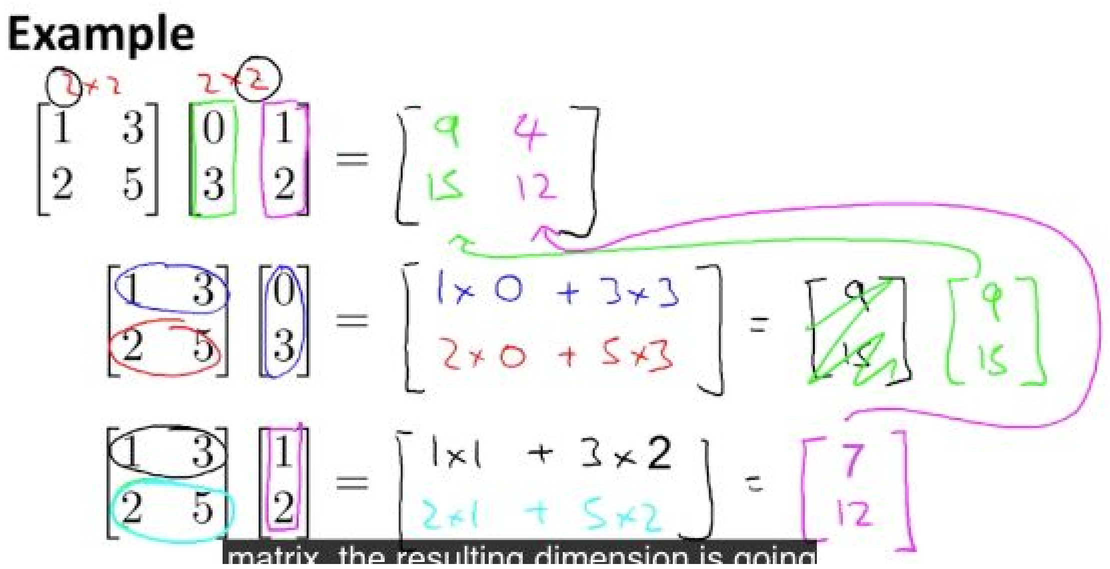
  - You multiply each element of the matrix by the scalar and the result is a new matrix of the same dimensions as the original matrix
- Matrix by vector multiplication
  - To get a result we multiply each line of the matrix by the vector and add the result up as shown below

    $$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a*x+b*y \\ c*x+d*y \\ e*x+f*y \end{bmatrix}$$

  - The result is a vector that has the number of rows of the original matrix and the number of columns of the vector (which, by default is 1)
- Matrix Matrix multiplication
  - To multiply matrices the number of rows in one matrix has to match the number of columns in the other matrix
  - The multiplications is broken down in cases of matrix vector multiplication with the resulting vectors being the columns in the result matrix as shown here:

    $$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} * \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} a*w+b*y & a*x+b*z \\ c*w+d*y & c*x+d*z \\ e*w+f*y & e*x+f*z \end{bmatrix}$$

  - And an actual example is



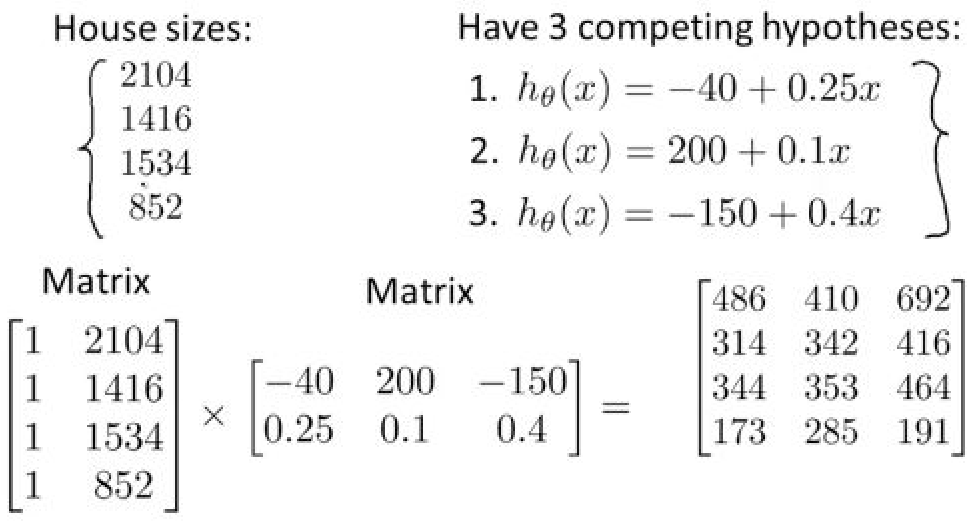
Example

$$\underset{2\times 2}{\begin{bmatrix}1 & 3\\2 & 5\end{bmatrix}}\underset{2\times 2}{\begin{bmatrix}0 & 1\\3 & 2\end{bmatrix}}=\begin{bmatrix}9 & 4\\15 & 12\end{bmatrix}$$

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

$$\begin{bmatrix}1 & 3\\2 & 5\end{bmatrix}\begin{bmatrix}1\\2\end{bmatrix}=\begin{bmatrix}1\times 1+3\times 2\\2\times 1+5\times 2\end{bmatrix}=\begin{bmatrix}7\\12\end{bmatrix}$$

- Matrix multiplication is used to apply multiple hypothesis to a set of values (i.e. house sizes) so we can predict the resulting price as shown below



House sizes:

2104
1416
1534
852

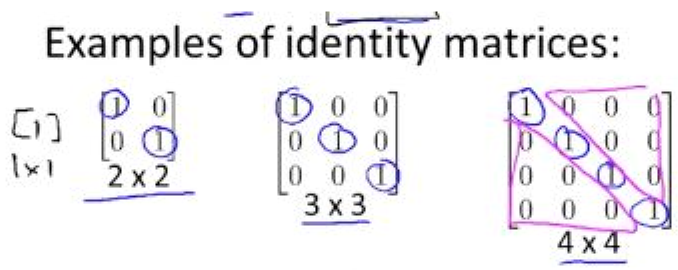
Have 3 competing hypotheses:

1. $h_\theta(x) = -40 + 0.25x$
2. $h_\theta(x) = 200 + 0.1x$
3. $h_\theta(x) = -150 + 0.4x$

Matrix Matrix

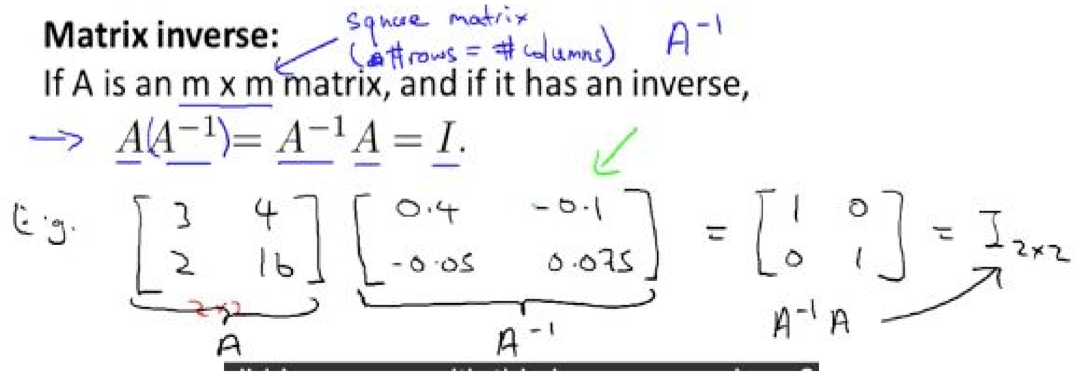
$$\begin{bmatrix}1 & 2104\\1 & 1416\\1 & 1534\\1 & 852\end{bmatrix}\times\begin{bmatrix}-40 & 200 & -150\\0.25 & 0.1 & 0.4\end{bmatrix}=\begin{bmatrix}486 & 410 & 692\\314 & 342 & 416\\344 & 353 & 464\\173 & 285 & 191\end{bmatrix}$$

- Matrix Multiplication Properties
  - Matrix multiplication is ***not commutative*** so A x B is not the same as B x A
  - Matrix multiplications ***is associative*** so if I have a 3 matrix multiplication, A x B x C I can solve it by doing (A x B) x C or by doing A x (B x C)
  - Identity Matrix
    - Identity matrix I is I_{nxn}
    - Examples are

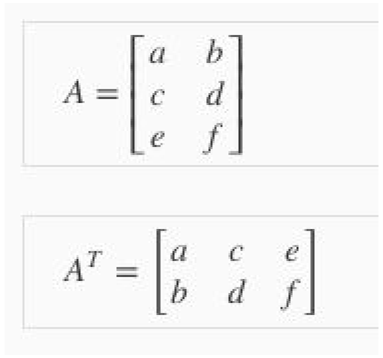


- Has 1's along the diagonals and 0 everywhere else
- A* I = I * A

- Inverse and Transpose Operations
  - Every number (except 0) has an inverse number so that number multiplied by its inverse gives you the identity (1), i.e. 12 multiplied by its inverse $\frac{1}{12}$ will give us 1
  - Matrix Inverse:
    - $A * A^{-1} = A^{-1} * A = I$
    - ***Only square matrices*** (a matrix where the number of rows = number of columns) ***have inverses***


Matrix inverse:
square matrix (# rows = # columns)
A^{-1}
If A is an m x m matrix, and if it has an inverse,
$A(A^{-1}) = A^{-1}A = I.$
e.g. $\underbrace{\begin{bmatrix} 3 & 4 \\ 2 & 16 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} 0.4 & -0.1 \\ -0.05 & 0.075 \end{bmatrix}}_{A^{-1}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I_{2\times2}$
2x2
$A^{-1}A$

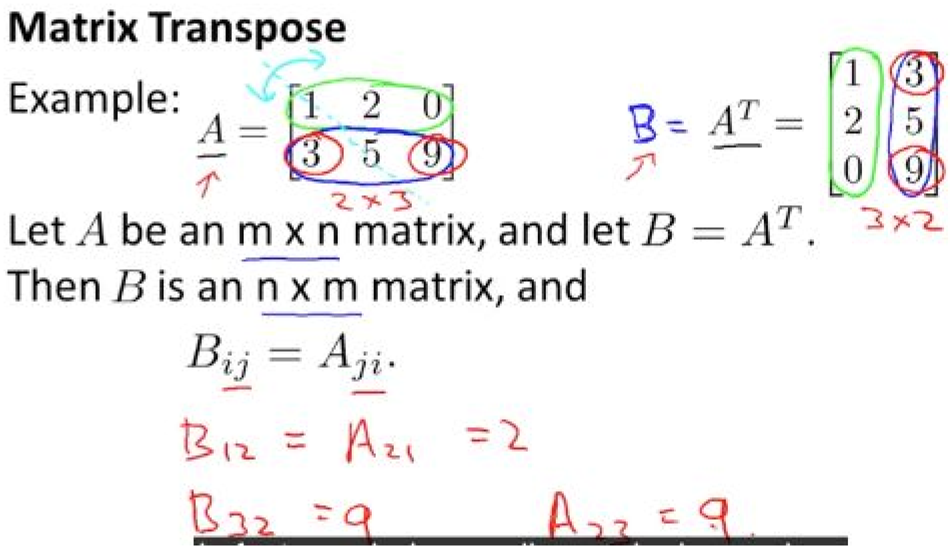
    - Matrices that don't have an inverse are called singular or degenerate
  - Matrix Transpose
    - The transposition of a matrix is like rotating the existing matrix 90 degrees in clockwise direction and reversing it



$$A = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$$

$$A^T = \begin{bmatrix} a & c & e \\ b & d & f \end{bmatrix}$$

    - An example is



Matrix Transpose

Example: $\underline{A} = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 5 & 9 \end{bmatrix}$ 2×3 $B = \underline{A^T} = \begin{bmatrix} 1 & 3 \\ 2 & 5 \\ 0 & 9 \end{bmatrix}$ 3×2

Let A be an m x n matrix, and let $B = A^T$.
Then B is an n x m matrix, and

$$B_{ij} = A_{ji}.$$

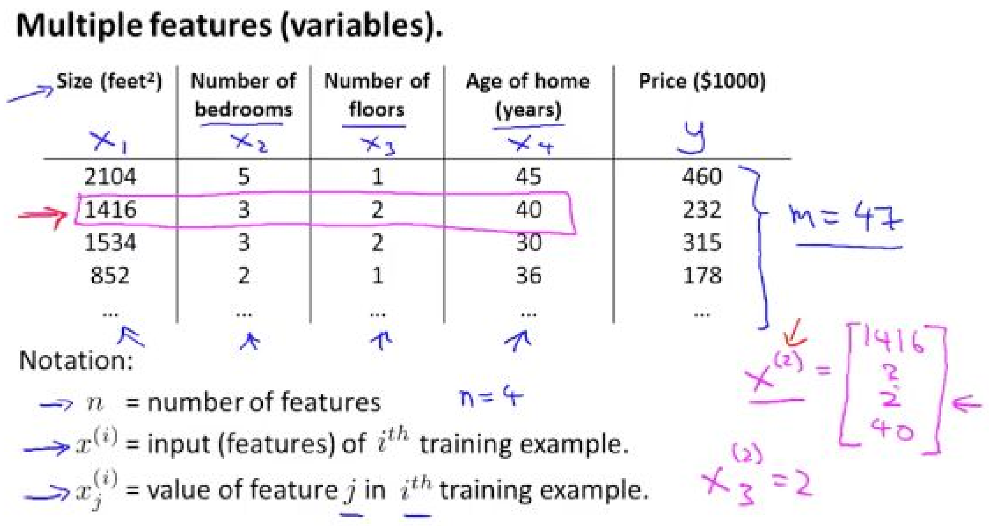
$B_{12} = A_{21} = 2$

$B_{32} = 9$ $A_{23} = 9$.

Week 2

Multiple Features

- Multiple features means you have more than one feature that you can use to predict the value. For example, to predict the price of a house (y) we originally had only one feature: the size in feet. Now we have the size in feet (x_1) + the number of rooms (x_2)+ the number of baths (x_3) + age of the home (x_4) as shown below



Multiple features (variables).

Size (feet²) x_1	Number of bedrooms x_2	Number of floors x_3	Age of home (years) x_4	Price ($1000) y
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
…	…	…	…	…

$m = 47$

Notation:

n = number of features $n = 4$

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

$x_j^{(i)}$ = value of feature j in i^{th} training example.

$x^{(2)} = \begin{bmatrix} 1416 \\ 3 \\ 2 \\ 40 \end{bmatrix}$

$x_3^{(2)} = 2$

- In this new world
  - n = number of features, in this case 4
  - m = number of rows/examples, in this case 47
  - x^i = index into my training example row. For example x^2 is the 2nd row in my training set which is a vector composed of the 4 values in row 2: [1416; 3; 2; 40]
  - x^i_j = the value of feature *j* in row *i*. For example x^2_3 means the value of the 3rd feature in the 2nd row of the above example, which is 2 so x^2_3 = 2
- Given the multiple variables our hypothesis function changes as follows:

Hypothesis:

Previously: $\underline{h_\theta(x) = \theta_0 \cancel{+ \theta_1 x}}$ $\nwarrow$

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4$$

$$\text{E.g. } \underline{h_\theta(x)} = \underline{80} + 0.1 x_1 + 0.01 x_2 + 3 x_3 - 2 x_4$$

The example above is saying the house starts with 80K price (θ_0) and then increases a little bit (0.1) with the size (θ_1). It continues to increase a little bit more (0.01) for each additional floor (θ_2) the house has. It increases more for each additional bedroom (θ_3) the house has and, finally, it loses some value for each age (θ_4) the house has

- We are changing a bit our hypothesis function by defining $x_0 = 0$. This creates an X vector that starts at 0 and we already had a θ vector that started at 0.

$$\rightarrow h_\theta(x) = \underline{\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n}$$

For convenience of notation, define $\boxed{x_0 = 1.}$ $\quad (x_0^{(i)} = 1)$

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

$$\underbrace{[\theta_0\ \theta_1 \cdots \theta_n]}_{\theta^T} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$$

$(n+1) \times 1$ matrix

$\theta^T x$ $\quad x$

$$h_\theta(x) = \underline{\overset{=1}{\theta_0 x_0} + \theta_1 x_1 + \cdots + \theta_n x_n}$$

$$= \boxed{\theta^T x.}$$

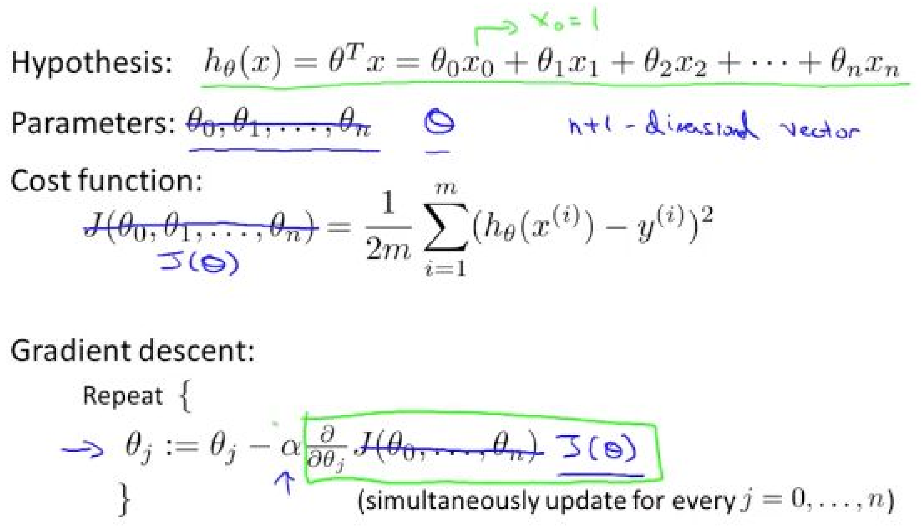
Multivariate linear regression. $\leftarrow$

Adding this definition allows to express our hypothesis as $\theta^T x$. Why? Because the original θ vector was an nx1 vector, its transpose becomes (n+1) x 1 vector, as shown above

- Finally, linear regression with multiple variables is also known as ***multivariate linear regression***

Gradient Descent with Multiple Variables

- For gradient descent we refer to the parameters as θ which is an n+1 dimensional vector so our cost function is now $J(\theta)$ and the gradient descent equation now multiplies the derivative * $J(\theta)$



Hypothesis: $h_\theta(x) = \theta^T x = \theta_0 \overset{x_0 = 1}{x_0} + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$

Parameters: ~~$\theta_0, \theta_1, \ldots, \theta_n$~~ θ n+1 - dimensional vector

Cost function:

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

Gradient descent:

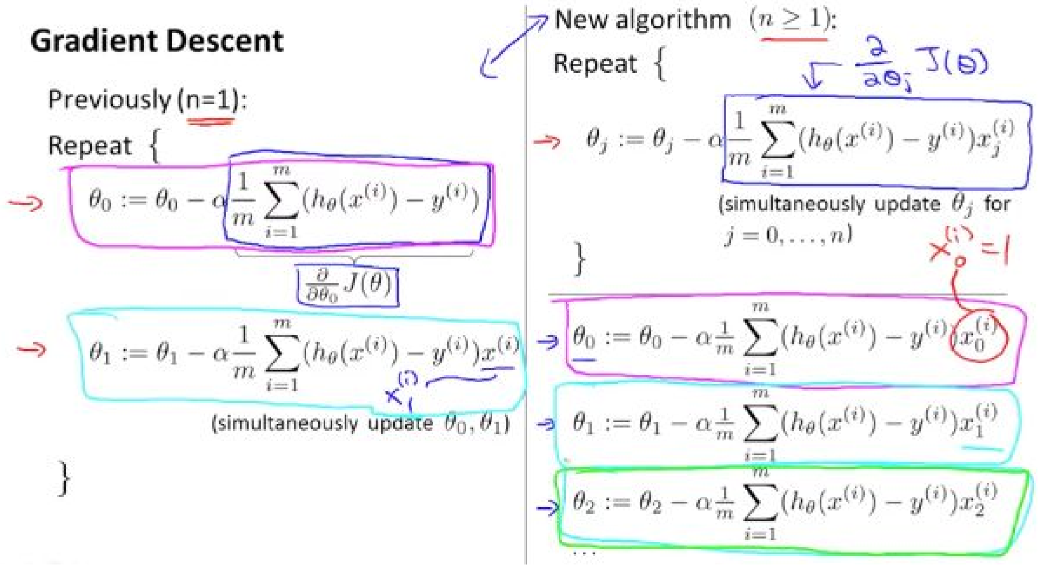
Repeat {

$$\theta_j := \theta_j - \alpha \boxed{\frac{\partial}{\partial \theta_j} \cancel{J(\theta_0, \ldots, \theta_n)} \; J(\theta)}$$

}

(simultaneously update for every $j = 0, \ldots, n$)

- Our equation is then modified as shown below. On the left side is the original gradient descent equation. On the right side is the new algorithm.



Gradient Descent

Previously (n=1):

Repeat {

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

$$\boxed{\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \underset{x_1^{(i)}}{x^{(i)}}}$$

(simultaneously update θ_0, θ_1)

}

New algorithm $(n \geq 1)$:

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

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

(simultaneously update θ_j for $j = 0, \ldots, n$)

}

$x_0^{(i)} = 1$

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

$$\boxed{\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_1^{(i)}}$$

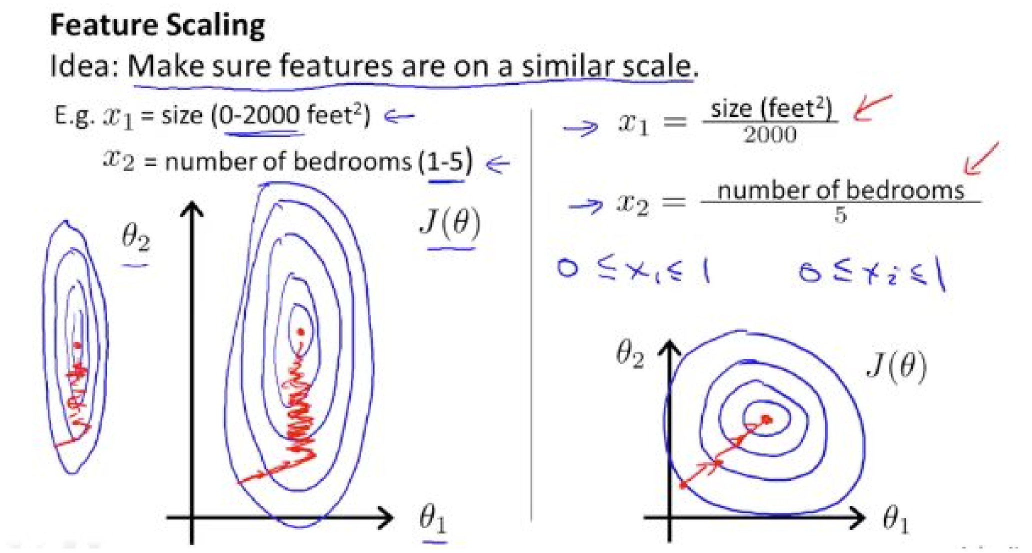
$$\boxed{\theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_2^{(i)}}$$

...

Notice how when we ‘unpack’ the gradient descent for multiple features equation we find that the equations for θ_0 and θ_1 are the same for their counterparts in the case of gradient descent for one variable

Gradient Descent in practice (Feature Scaling & Mean Normalization)

- ***Feature Scaling*** is the idea of having the features in a hypothesis to take on a similar scale to speed up the convergence of gradient descent. The left side below shows that having 2 features with widely different scale will result in having big skinny ellipses which will make gradient descent take a long time to converge. The solution to that is to ‘scale’ both features (diving house size/2000 and bedrooms/5) which will make the feature range of values between 0 and 1. This will result in smaller, fatter ellipses which will make grade descent converge faster.



Feature Scaling

Idea: Make sure features are on a similar scale.

E.g. x_1 = size (0-2000 feet^2)

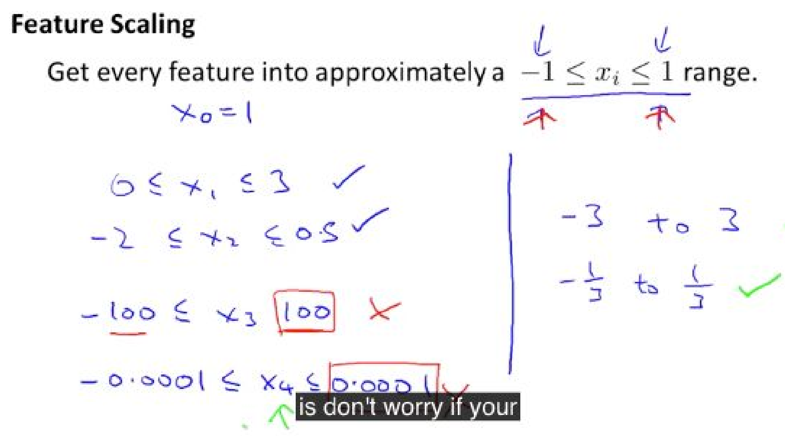
x_2 = number of bedrooms (1-5)

$x_1 = \frac{\text{size (feet}^2)}{2000}$

$x_2 = \frac{\text{number of bedrooms}}{5}$

$0 \leq x_1 \leq 1 \qquad 0 \leq x_2 \leq 1$

- Generalizing, what we are trying to do is to get every feature between -1 and 1. We don’t worry if we are not exactly there but are in the neighborhood as shown by the examples below



Feature Scaling

Get every feature into approximately a $-1 \leq x_i \leq 1$ range.

$x_0 = 1$

$0 \leq x_1 \leq 3$ ✓

$-2 \leq x_2 \leq 0.5$ ✓

$-100 \leq x_3 \boxed{100}$ ✗

$-0.0001 \leq x_4 \leq \boxed{0.0001}$ ✗

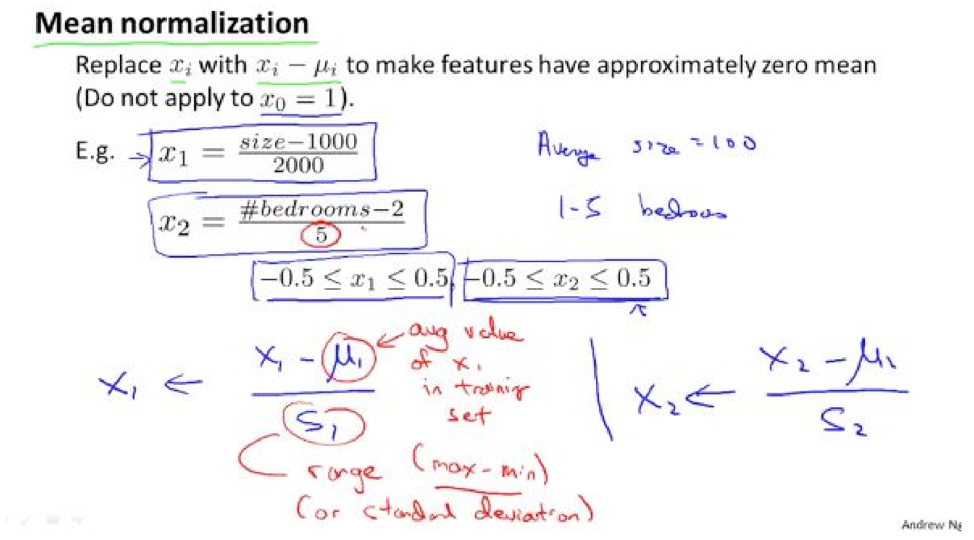
-3 to 3 ✓

$-\frac{1}{3}$ to $\frac{1}{3}$ ✓

is don't worry if your

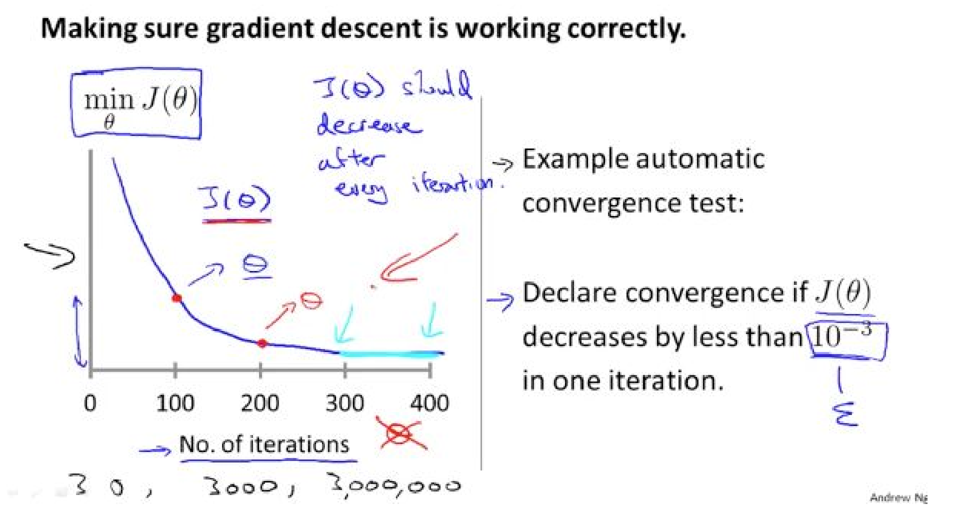
In the example above x_1 and x_2 are OK since their range is close to the desired range. x_3 and x_4 are considered poorly scaled since their range is not close to the desired range

- Another thing to do is ***mean normalization***. Mean normalization makes the values of each feature to have zero mean. We achieve this by taking the original feature value, finding the average and we then do the following formula: $x_1 = \frac{feature - average}{max - min}$ where $max - min$ is the standard deviation. In a more formal way we can write $x_i = \frac{x_i - \mu_i}{\varsigma_i}$. We do this to make sure gradient descent converges faster


Mean normalization
Replace x_i with $x_i - \mu_i$ to make features have approximately zero mean
(Do not apply to $x_0 = 1$).
E.g. $x_1 = \frac{size-1000}{2000}$
$x_2 = \frac{\#bedrooms-2}{5}$
$-0.5 \le x_1 \le 0.5$ $-0.5 \le x_2 \le 0.5$
Average size = 100
1-5 bedrooms
$x_1 \leftarrow \frac{x_1 - \mu_1}{S_1}$
avg value of x_1 in training set
range (max - min)
(or standard deviation)
$x_2 \leftarrow \frac{x_2 - \mu_1}{S_2}$
Andrew Ng

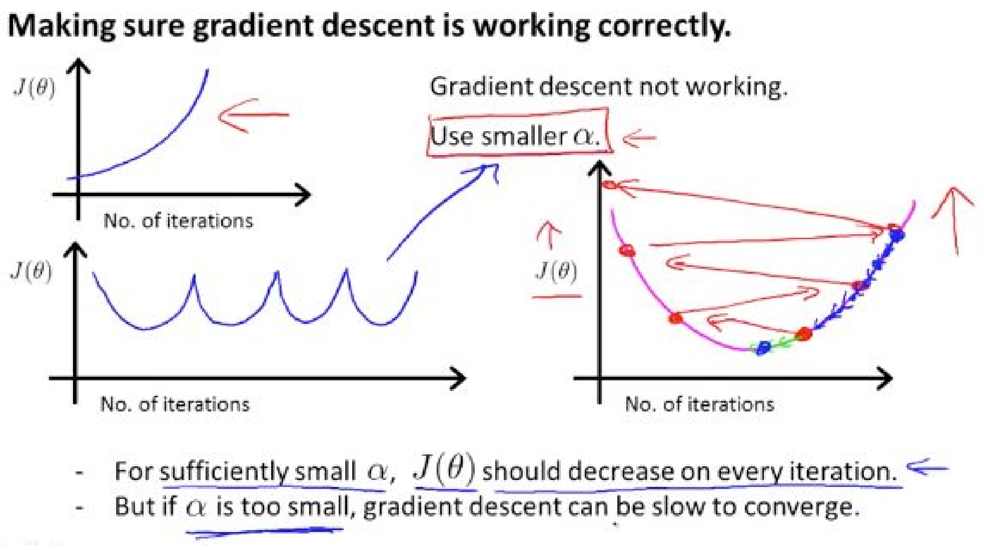
Gradient Descent in practice (Learning Rate)

- A suggestion is to plot a graph showing number of iterations against the cost function. This graph can show us if gradient descent is working and can also tells us when we are done (i.e we can declare convergence if the cost function value is not changing significantly from one iteration to the next)



Another way is to declare convergence if the cost function decreases by less than a small value in one iteration. Professor recommends using the plots since gradient descent converges at different iterations depending on the function

- Gradient descent is not working if the cost function $J(\theta)$ increases as the number of iterations increase. The solution is to use a smaller α. If $J(\theta)$ increase and decreases as shown in the 2nd chart below it means we also need to use a smaller α

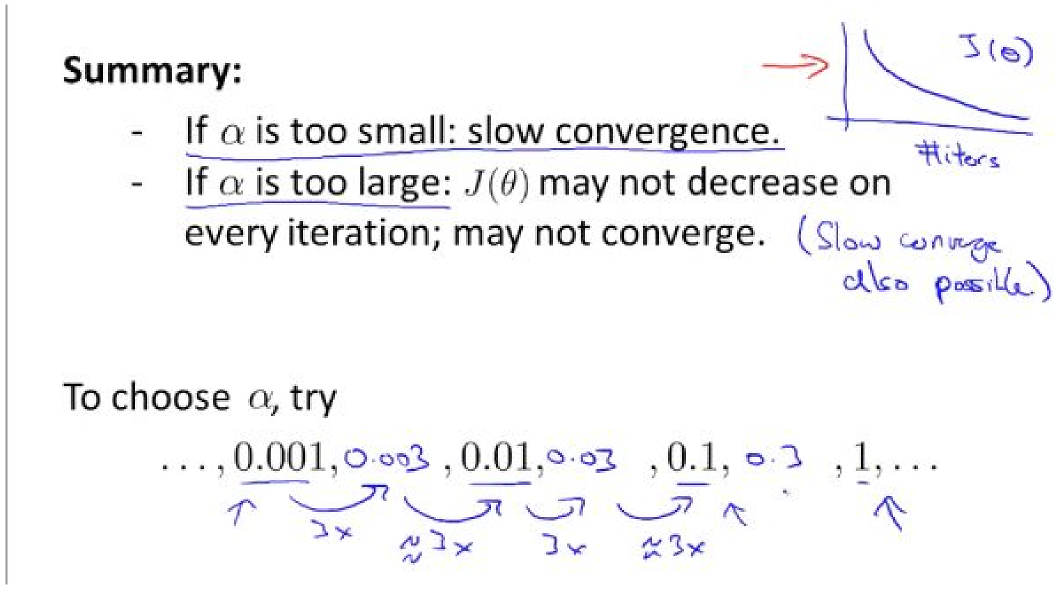


Making sure gradient descent is working correctly.

- For sufficiently small α, $J(\theta)$ should decrease on every iteration.
- But if α is too small, gradient descent can be slow to converge.

mathematicians have proven that for a small enough α, $J(\theta)$ should decrease in every iteration. However, if α is too small, gradient descent can be slow to converge

- Summarizing, if α is too small, gradient descent will be slow to converge. If it is too large cost function not decrease on every iteration or will be slow to converge.



Summary:

- If α is too small: slow convergence.
- If α is too large: $J(\theta)$ may not decrease on every iteration; may not converge. (Slow converge also possible)

To choose α, try

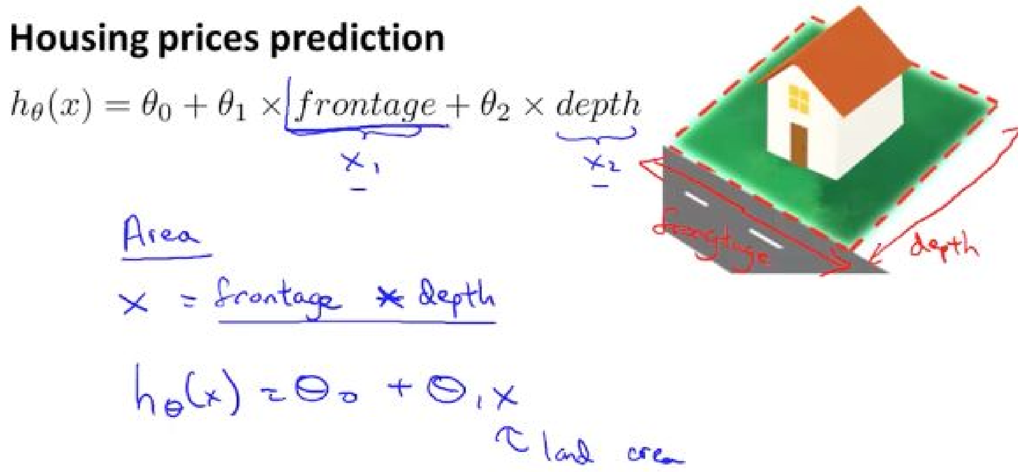
$$\ldots, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, \ldots$$

$J(\theta)$
#iters
3x
≈3x
3x
≈3x

To choose α try a range of values increasing each value roughly 3 times and this should usually work

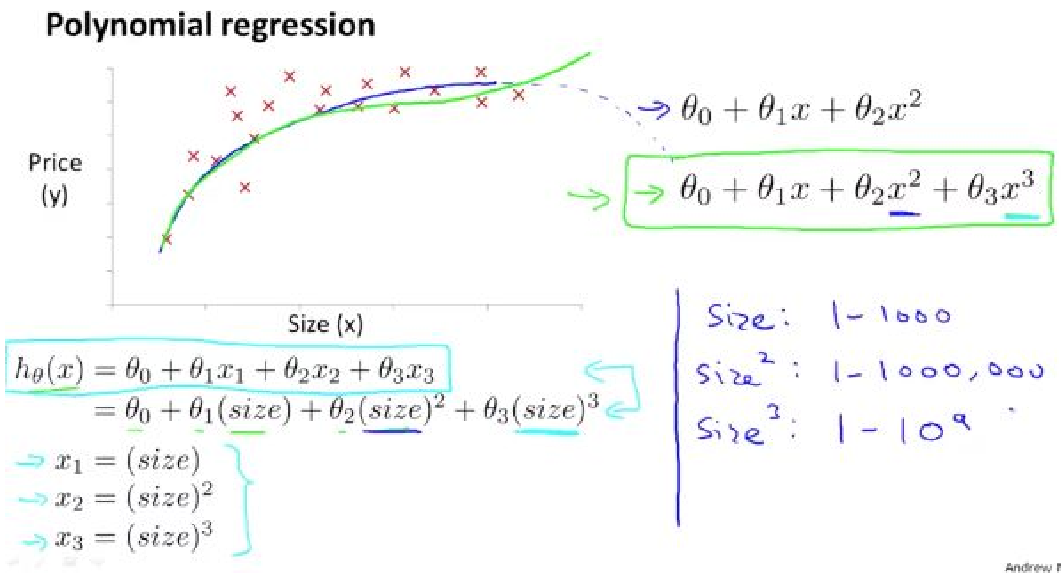
Features & Polynomial Regression

- Sometimes we can create 'new' features from the features we are given so we can have a better model as shown in the example below. In this example we are given the frontage and depth of the house and we create a new feature called area (x) and that is the feature we use in the model



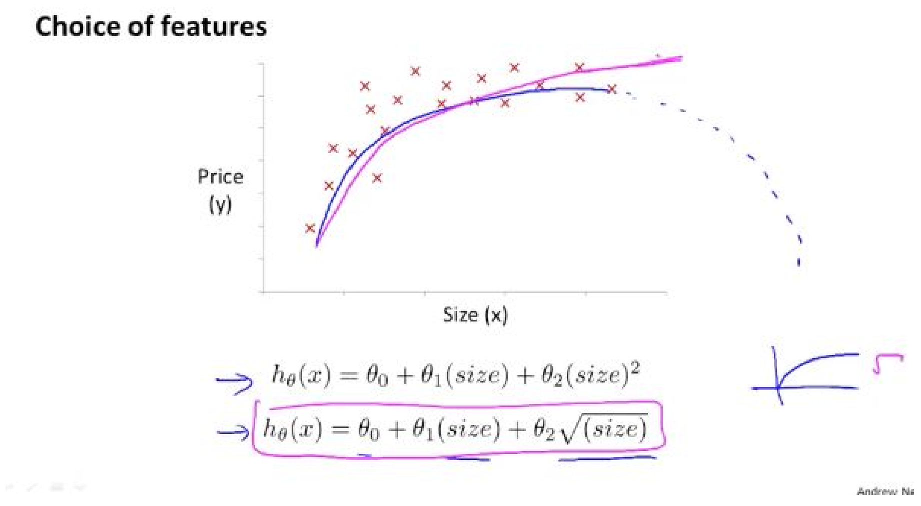
Housing prices prediction
$h_\theta(x) = \theta_0 + \theta_1 \times \underbrace{frontage}_{x_1} + \theta_2 \times \underbrace{depth}_{x_2}$
Area
x = frontage * depth
$h_\theta(x) = \theta_0 + \theta_1 x$
land area
frontage
depth

- Polynomial Regression is the idea of using a polynomial function as the equation for our hypothesis as shown below. A polynomial equation might be a better fit depending on the data and sometimes you might have to go into a cubic equation. In such cases each x parameter might have a widely different range so feature scaling becomes more important here



In this example we continue with the size of the house as our parameter but we add the square of the size and the cube of the house to get a polynomial of the 3rd degree. A quadratic function does not fit the above data because it eventually comes down and we expect our house prices to remain constant or increase as the size of the house grows

- Another reasonable choice is to use the square root function which seems to be a good fit for the data in the example below. In this case, a quadratic function does not work again and we turn to the square root of the size to see if that parameter gives us a better model



$$h_\theta(x) = \theta_0 + \theta_1(size) + \theta_2(size)^2$$

$$\boxed{h_\theta(x) = \theta_0 + \theta_1(size) + \theta_2\sqrt{(size)}}$$

Normal Equation for Multivariate Linear Regression

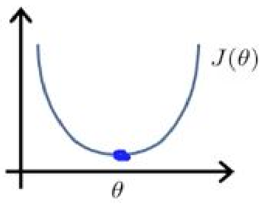
- Normal Equation is an alternative to the gradient descent algorithm we have been using for linear regression
- Normal equation will allow us to solve for the optimal value of θ in one step instead of doing the iterative steps of gradient descent
- The normal equation will use the partial derivative of $J(\theta)$ for every J and set it equal to zero and then solve of every θ .

Intuition: If 1D $(\theta \in \mathbb{R})$

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

$$\frac{\partial}{\partial\theta} J(\theta) = \dots \overset{set}{=} 0$$

Solve for θ



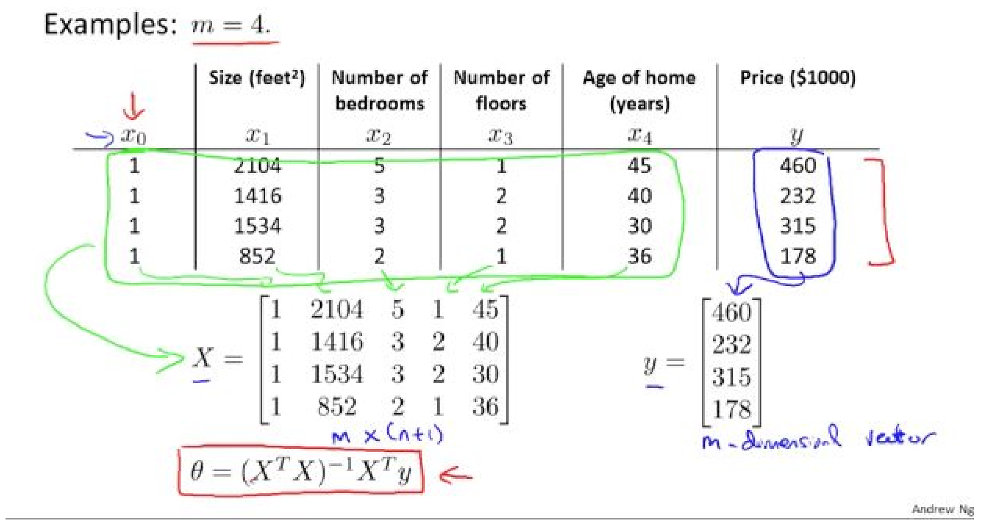
$$\theta \in \mathbb{R}^{n+1} \qquad J(\theta_0, \theta_1, \dots, \theta_m) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$\frac{\partial}{\partial\theta_j} J(\theta) = \cdots \overset{set}{=} 0 \qquad \text{(for every } j\text{)}$$

Solve for $\theta_0, \theta_1, \dots, \theta_n$

Andrew Ng

- In the example below the training set (m=4) becomes a matrix X of m x n+1 where n is the number of parameters. The y (prices in this case) becomes an m-dimensional vector. For this case we can find θ with the equation at the bottom: $\theta = (X^T X)^{-1} X^T y$



Examples: $m = 4.$

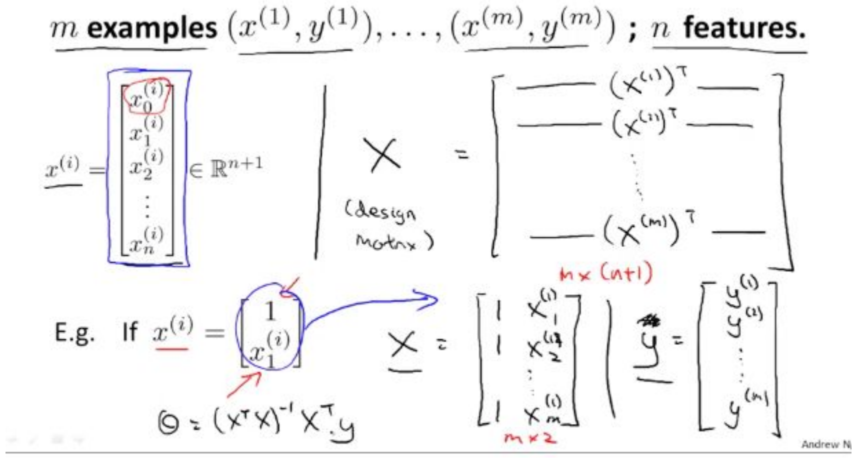
x_0	Size (feet²) x_1	Number of bedrooms x_2	Number of floors x_3	Age of home (years) x_4	Price ($1000) y
1	2104	5	1	45	460
1	1416	3	2	40	232
1	1534	3	2	30	315
1	852	2	1	36	178

$$X = \underbrace{\begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{bmatrix}}_{m \times (n+1)} \qquad y = \underbrace{\begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}}_{m\text{-dimensional vector}}$$

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

Andrew Ng

- In a more general way, having m examples and n features each $x^{(i)}$ is a vector and we take its transpose to create the X matrix, also known as the design matrix. Y becomes an m dimensional vector and the equation to find theta is $\theta = (X^T X)^{-1} X^T y$



m **examples** $(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})$; n **features.**

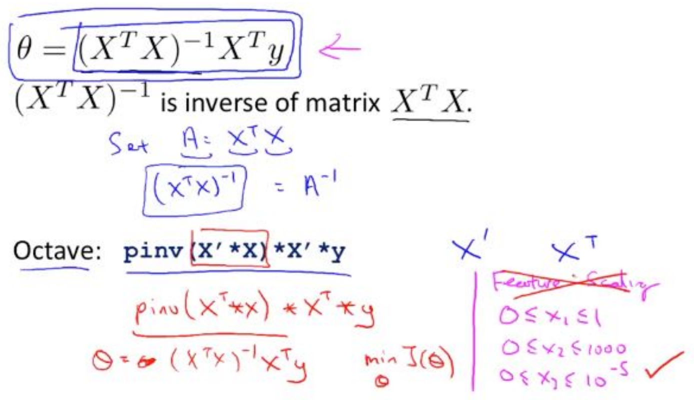
$$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix} \in \mathbb{R}^{n+1} \qquad \underset{\text{(design matrix)}}{X} = \underbrace{\begin{bmatrix} \text{---} & (x^{(1)})^T & \text{---} \\ \text{---} & (x^{(2)})^T & \text{---} \\ & \vdots & \\ \text{---} & (x^{(m)})^T & \text{---} \end{bmatrix}}_{m \times (n+1)}$$

E.g. If $x^{(i)} = \begin{bmatrix} 1 \\ x_1^{(i)} \end{bmatrix}$ $\quad X = \underbrace{\begin{bmatrix} 1 & x_1^{(1)} \\ 1 & x_2^{(1)} \\ \vdots & \vdots \\ 1 & x_m^{(1)} \end{bmatrix}}_{m \times 2} \qquad y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$

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

Andrew Ng

- The first term of a normal equation is the inverse of the multiplication of $X^T X$. We then multiply that by X^T and then by y. The octave command is shown below. Finally, when using normal equation you don't need to do feature scaling. Feature scaling is still needed for Gradient Descent.



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

$(X^T X)^{-1}$ is inverse of matrix $X^T X$.

Set $A = X^T X$

$\boxed{(X^T X)^{-1}} = A^{-1}$

Octave: `pinv(X'*X)*X'*y`

$\text{pinv}(X^T * X) * X^T * y$

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

X' $\quad X^T$

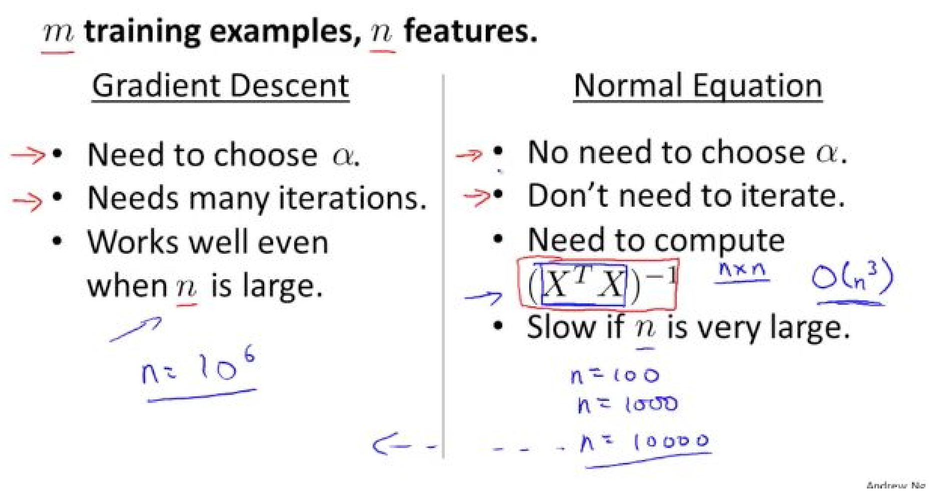
~~Feature Scaling~~

$0 \le x_1 \le 1$

$0 \le x_2 \le 1000$

$0 \le x_3 \le 10^{-5}$ ✓

- When to use gradient descent and when to use normal equation? If n is larger than 10,000 I would start considering gradient descent because, computationally speaking, inverting a 10K x 10K matrix starts to get slow.

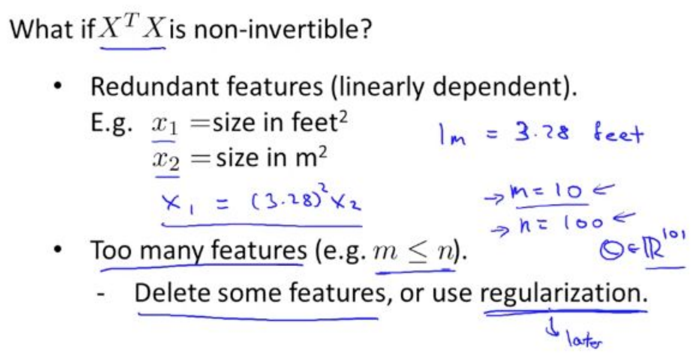


m training examples, n features.

Gradient Descent	Normal Equation
→ • Need to choose α.	→ • No need to choose α.
→ • Needs many iterations.	→ • Don't need to iterate.
• Works well even when n is large.	• Need to compute → $(X^T X)^{-1}$ $\quad n \times n$ $\quad O(n^3)$
↗ $n = 10^6$	• Slow if n is very large.
← -	$n = 100$
	$n = 1000$
	- - - $n = 10000$

Normal Equation and Non invertibility

- Sometimes $X^T X$ cannot be inverted, what do we do? If we use pinv in octave we are fine, it will do the right thing
- What are the causes why $X^T X$ is non invertible?
  - Look at features to see if we have redundant features as shown below and delete one of the features, you don't need both. Or
  - Do you have too many features? If so, delete some features or use regularization (which we will talk about it later in this class).



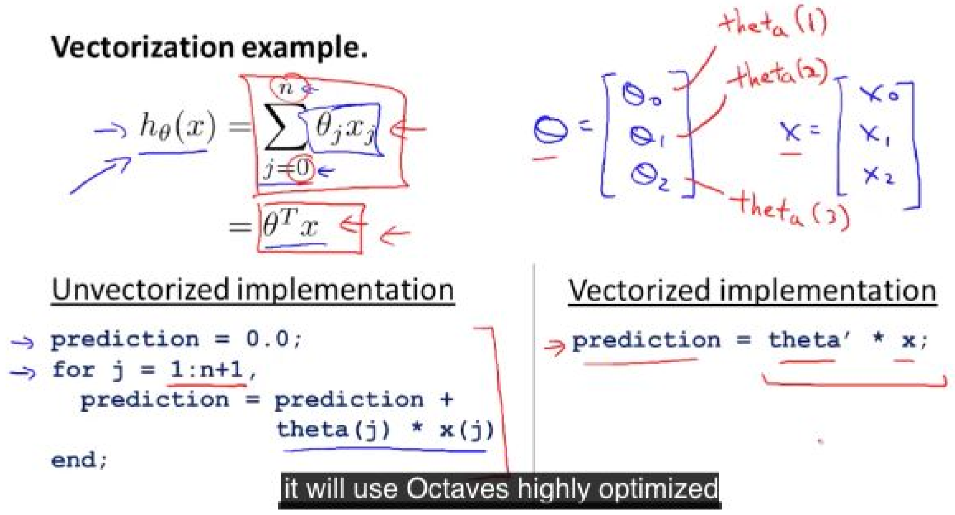
What if $X^T X$ is non-invertible?

- Redundant features (linearly dependent).
  E.g. x_1 = size in feet^2
  x_2 = size in m^2
  $x_1 = (3.28)^2 x_2$
- Too many features (e.g. $m \leq n$).
  - Delete some features, or use regularization.

$1m = 3.28$ feet
$\rightarrow m = 10 \leftarrow$
$\rightarrow n = 100 \leftarrow$
$\theta \in \mathbb{R}^{101}$
later

Octave Review

- We will use Octave as the program to do homework on.
- Vectorization
  - We will let Octave do a lot of the work for us. For example, for linear regression we could have a non vectorized implementation (which is more lines of code) or we can have a vectorized implementation with 1 line of code as shown below



Vectorization example.

$$h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j$$
$$= \theta^T x$$

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix} \quad x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

Unvectorized implementation

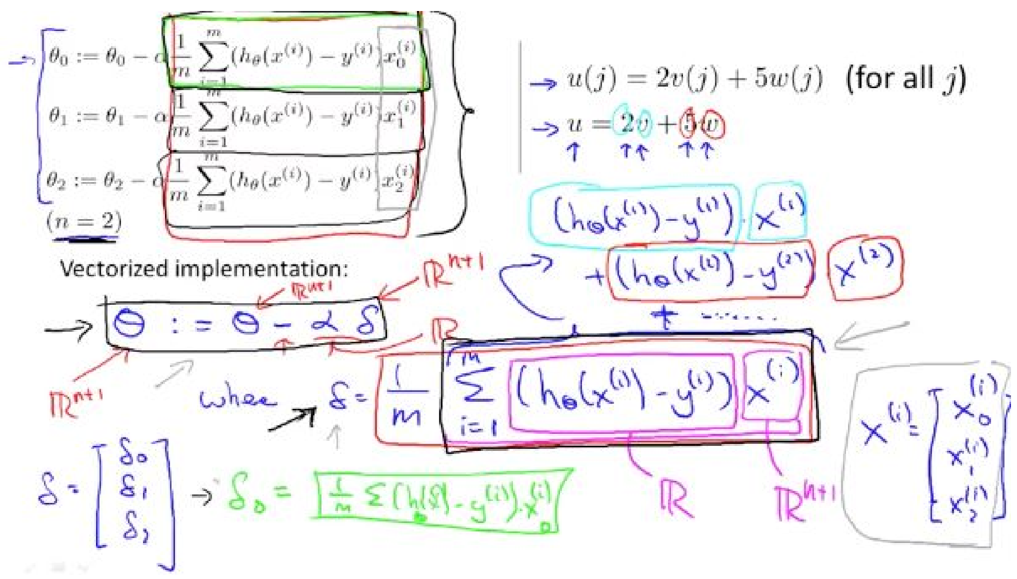
```
prediction = 0.0;
for j = 1:n+1,
  prediction = prediction +
               theta(j) * x(j)
end;
```

Vectorized implementation

```
prediction = theta' * x;
```

it will use Octaves highly optimized

  We can think of linear regression as multiplying θ^T times our x and that results in one line of code. This line of code will run more efficiently than the example of the left since it uses the highly optimized ML routines Octave has
  - Gradient Descent vectorized implementation for Linear Regression



$\theta_0 := \theta_0 - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}$
$\theta_1 := \theta_1 - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_1^{(i)}$
$\theta_2 := \theta_2 - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_2^{(i)}$
$(n = 2)$
$u(j) = 2v(j) + 5w(j)$ (for all j)
$u = 2v + 5w$
Vectorized implementation:
$\Theta := \Theta - \alpha\delta$
$\mathbb{R}^{n+1}$
where $\delta = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x^{(i)}$
$(h_\theta(x^{(1)}) - y^{(1)})x^{(1)} + (h_\theta(x^{(2)}) - y^{(2)})x^{(2)} + \cdots$
$\mathbb{R}$
$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \end{bmatrix}$
$\delta = \begin{bmatrix} \delta_0 \\ \delta_1 \\ \delta_2 \end{bmatrix} \rightarrow \delta_0 = \frac{1}{m}\sum(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}$

We look at this equation as the addition or subtraction of n vectors. $\theta = \theta - \alpha\delta$ is the equation where $\delta = \frac{1}{m}\sum_{i=1}^{m}(h_0(x_i) - y_i) * x^i$

Linear Regression Programming Assignment

- There are 2 main questions in this assignment: Computing Cost Function & Gradient Descent
- Computing Cost function
  - We have to compute the cost function using the usual equation: $\frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)} - y^{(i)})^2$
  - $h_\theta = X * \theta$ where X is the a m x 2 (in this case) matrix that has all the samples (including θ_0 which is represented as a column of 1's, which is why the X has 2 columns, 1 column for θ_0 and the other column for θ_1 (the population size)) and θ is a 2 x 1 vector since there are only 2 theta values. You then do the multiplication which is multiplying X (m x 2) * θ (2 x 1) and the resulting matrix is of dimensions m x 1. The line of code is *h = X * theta*
  - We then subtract y from h, y is an m x 1 vector that has all the predictions so its dimensions are m x 1 so we subtract h (m x 1) - y (m x 1) and get the error. The line of code is *error = h - y*
  - Now we need the square which is done like this: *error_sqr = error.*2*;
  - Finally, we do the sum and multiply by 1/2m as follows: *J = 1/(2*m) * sum(error_sqr)*;
  - All lines of code are

```
h= X * theta;
 error = h - y;
```

error_sqr = error.^2;
*J= 1/(2*m) * sum(error_sqr);*

- Gradient Descent:
  - in this case what we want is θ, whose equation is for each iteration is $\theta = \theta - \alpha * \frac{1}{m}(\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})$ where $h_\theta(x^{(i)})$ = X * θ, X is an m x n matrix that has all the training set rows (in the example it's 2 rows, one for θ_0 that has all 1's and one for the population sizes) and θ is an *n x 1* vector that has the 2 parameters. These 2 matrices can be multiplied and the result *h* is a matrix of dimension m x 1 (97 x 1 in this case). The line of code is *h = X * theta;*
  - Now that we have h we subtract y from it (both have dimensions m x 1) to get the error: *error = h - y;*
  - Then we get the total equation by multiplying $\alpha * \frac{1}{m}(\sum_{i=1}^{m} X \ * \ error)$ However, note that the vector multiplication automatically includes the sum of the products so our line of code should not include a sum(). Additionally, note that at the end we want a result of n x 1 dimension (like θ) and error is of dimension *m x 1* but X is of dimension *m x n* so we need to transpose X so it can have a dimension of n x m; that way our matrix multiplication)X' (*n x m*) * error (*m x 1*) can happen and at the end I have an *nx1* vector). Therefore the line of code is: *theta_change = alpha * 1/m * (X' * error)*
  - All lines of code are:
    *h = X * theta;*
    error = h - y;
    *theta_change = alpha * (1/m) * (X' * error);*
    theta = theta - theta_change;
- Linear Regression with multiple variables (optional)
  - Feature Normalization: we take X (a matrix of *m x n* dimensions, it contains all the training sample) and we have to normalize it as follows: $\frac{X_i - \mu_i}{\varsigma_i}$. In practice this means we have to
    - find the mean $mu = mean(X);$
    - find the standard deviation: $sigma \ = \ std(X);$
    - The mean and sigma have to be subtracted and divided for every element in the matrix so we need to create a mu_matrix and a sigma_matrix of the same size as X so we can do the operations
      - find the size of X: $m \ = \ size(X, 1);$ X at this point is 47 x 2
      - create a matrix mu starting with a column of ones and then multiplying by mu (we end up with a 47 x 2 matrix): $mu_{matrix} = ones(1, m) \ * \ mu;$
      - create a matrix sigma: $sigma_{matrix} \ = \ ones(1, m) \ * sigma;$ same dimensions as mu (47 x 2)

        - Then we can do the operations:
            - X_norm = X - mu_matrix;
            - X_norm = X_norm./ sigma_matrix;
            - The resulting 47 x 2 matrix is now normalized and θ_0 is added later to make it a 47 x 3 matrix
        - Final code is then:

```
mu = mean(X);  % mean for the whole training set
sigma = std(X); % standard deviation for the whole training set

m = size(X, 1);  %number of rows, 47 in this case
mu_matrix = ones(m,1) * mu;  % result is a 47 x 2 matrix
sigma_matrix = ones(m,1) * sigma; % result is a 47 x 2 matrix

X_norm = X - mu_matrix;
X_norm = X_norm./ sigma_matrix;
```

    - Gradient descent: here we apply gradient descent:
        - Hypothesis h = X * theta;
        - Error = h - y;
        - Theta_change = alpha * (1/m) * (X' * error)
        - Theta = theta - theat_change;
    - We are then asked to do a prediction for a particular case: a house with 1650 square feet and 3 bedrooms. For that we do the following lines of code:

```
z = [1650; 3];           % vector with the values we will use
z_norm = z' - mu;  % normalizing z by taking the transpose and subtracting mu
z_norm = z_norm./ sigma;  %now we divide by sigman and our vector is now scaled
z_final = [1 z_norm]; % add a 1 so we can multiply by theta
price = z_final * theta;
```

    - Normal Equations:
        - no iterations, just one calculation: $\theta = (X^T X)^{-1} X^T y$
        - and in octave the line of code is *theta = pinv(X' * X) * X' * y;*
    - Finally, we do a prediction for the same case (a house with 1,650 square feet and 3 bedrooms). Normal equations do not need scaling so no need to normalize
        - We define a vector with the 3 values and
        - then we multiply theta' by the vector

```
z = [1; 1650; 3];
price = theta' * z;
```

At the end of the day we should remember:

- *In linear regression*
    - *the hypothesis h =* $\theta^T * X$
    - *The cost function is 1/(2m) ** $\sum_{i=1}^{m} (h^{(i)}_\theta - y^{(i)})^2$ *where y is the vector with the results*

- *Gradient descent equation for each theta change is* $\alpha * 1/m * (X' * error)$ *where* $h = X * \theta$ *and* $error = h - y$*. You have an initial theta and each calculated iteration of theta is subtracted from the original theta*
- *Feature scaling (used to make gradient descent converge faster) means you need to calculate the mean and sigma and subtract mean for every value of X and divide every value of X by sigma*
- *Normal equation calculation is* $\theta = (X^T X)^{-1} X^T y$ *and the corresponding octave code is pinv(X' * X) * X' * y. Normal equation does not need feature scaling*
- *For one particular prediction you apply the hypothesis equation (*$\theta^T * X$*) to the relevant values*