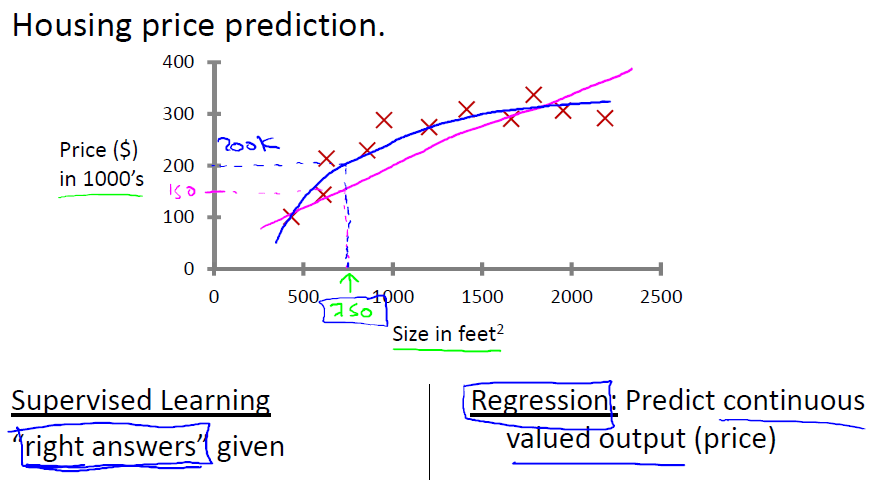
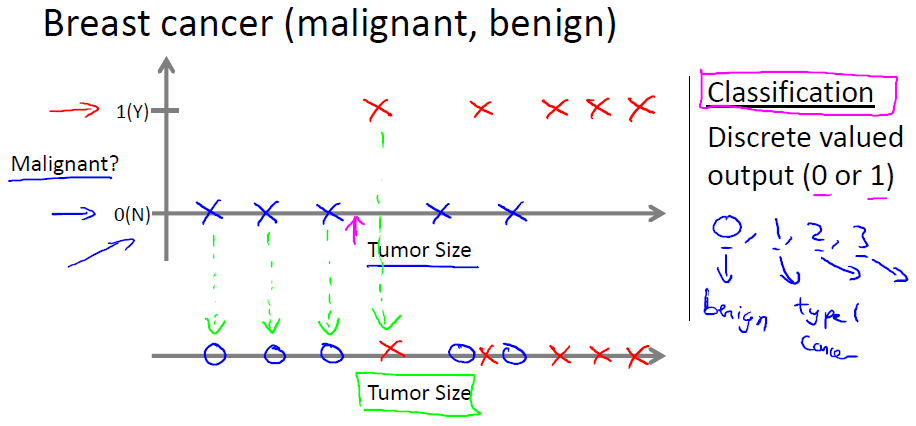
**Machine learning** – a computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E, e.g. T – the task of playing checkers, E – the experience od playing many games, P – the probability that the program will win the next game.

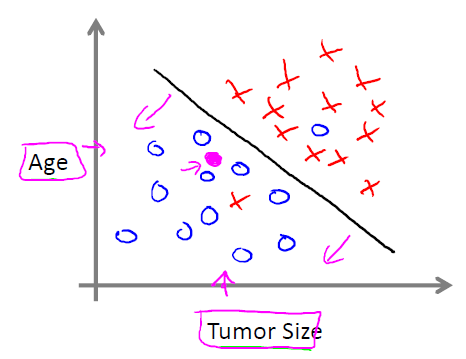
**Two classifications of ML:**

* Supervised learning – we are given a data set and already know what our correct output should look like, having the idea that there is a relationship between the input and the output.
  + Regression problems – we are trying to predict results within a continuous output, meaning that we are trying to map input variables to some continuous function, e.g. given data about the size of houses on the real estate market, try to predict their price; given a picture of a person, we have to predict their age

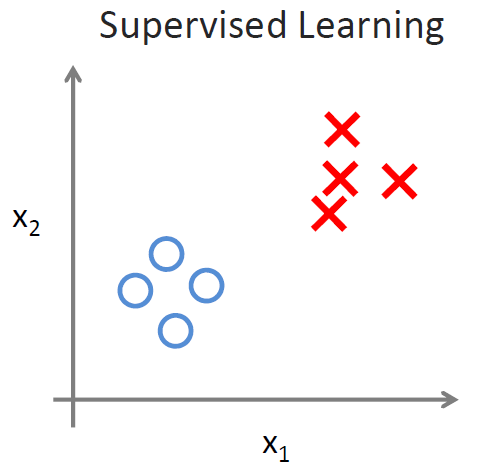
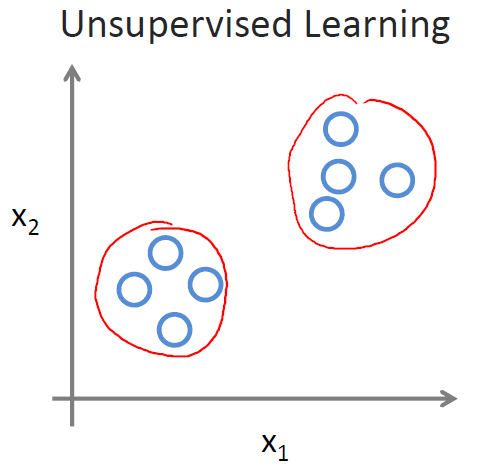


* + Classification problems – we are trying to predict results in a discrete output. In other words, we are trying to map input variables into discrete categories, e.g. given a patient with a tumor, we have to predict whether the tumor is malignant or benign





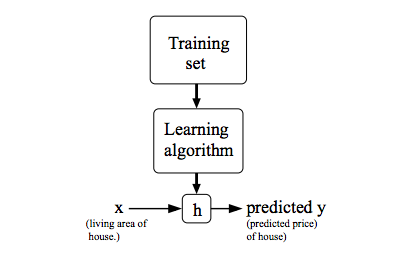
* Unsupervised learning – allows us to approach problems with little or no idea what our results should look like. We can derive structure from data where we don't necessarily know the effect of the variables. We can derive this structure by clustering the data based on relationships among the variables in the data. With unsupervised learning there is no feedback based on the prediction results.

* + Clustering – e.g. take a collection of 1,000,000 different genes, and find a way to automatically group these genes into groups that are somehow similar or related by different variables, such as lifespan, location, roles, and so on.
  + Non-clustering – e.g. the "Cocktail Party Algorithm", allows you to find structure in a chaotic environment. (i.e. identifying individual voices and music from a mesh of sounds at a coctail party).

**Model representation:**

* x(i) – used to denote the “input” variables, also called input features (e.g. a size of the house)
* y(i) – used to denote the “output” or target variable that we are trying to predict (e.g. a price of the house)
* a pair (x(i),y(i)) is called a training example, and the dataset that we’ll be using to learn — a list of m training examples (x(i), y(i)); i = 1, ..., m — is called a training set
* X – used to denote the space of input values
* Y – used to denote the space of output values
* Linear regression with one variable is also known as "univariate linear regression." Univariate linear regression is used when you want to predict a **single output** value y from a **single input** value x.
* In supervised learning – our goal is, given a training set, to learn a function h : X → Y so that h(x) is a “good” predictor for the corresponding value of y. This function h is called a hypothesis. Seen pictorially, the process is therefore like this:



* Hypothesis function – we give to hθ(x) values for θ0 and θ1 to get our estimated output y^. In other words, we are trying to create a function called hθ that is trying to map our input data (the x's) to our output data (the y's). We will be trying out various values of θ0 and θ1 to try to find values which provide the best possible "fit" or the most representative "straight line" through the data points mapped on the x-y plane.

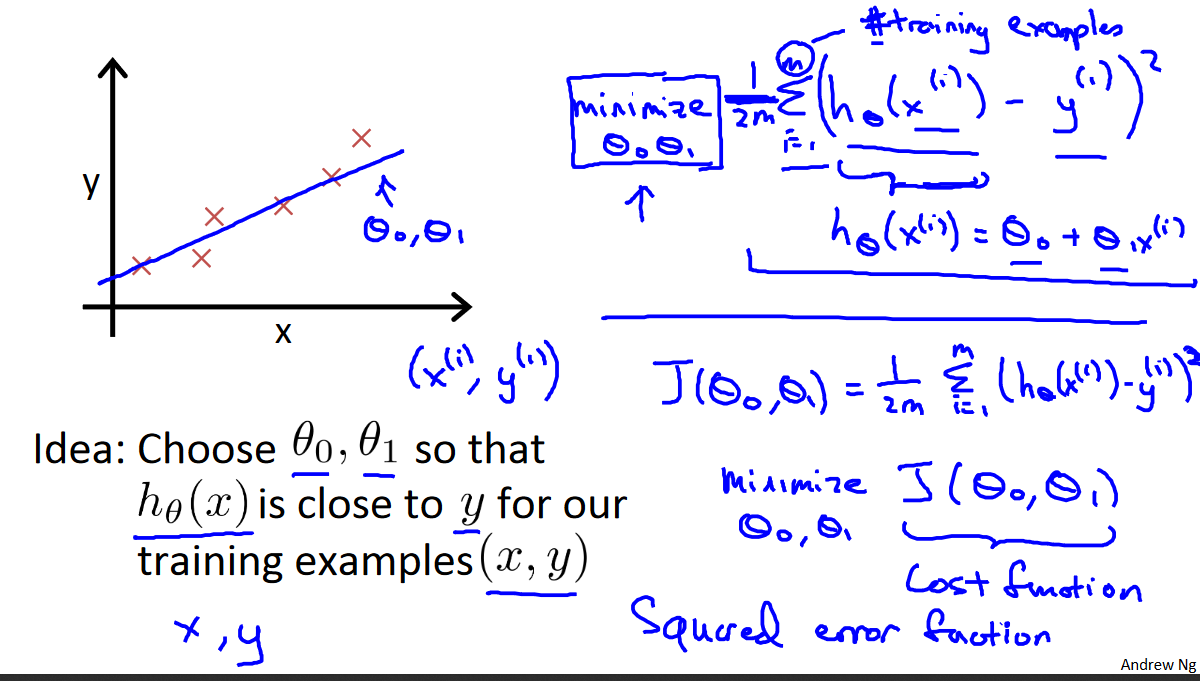


**Cost function:**

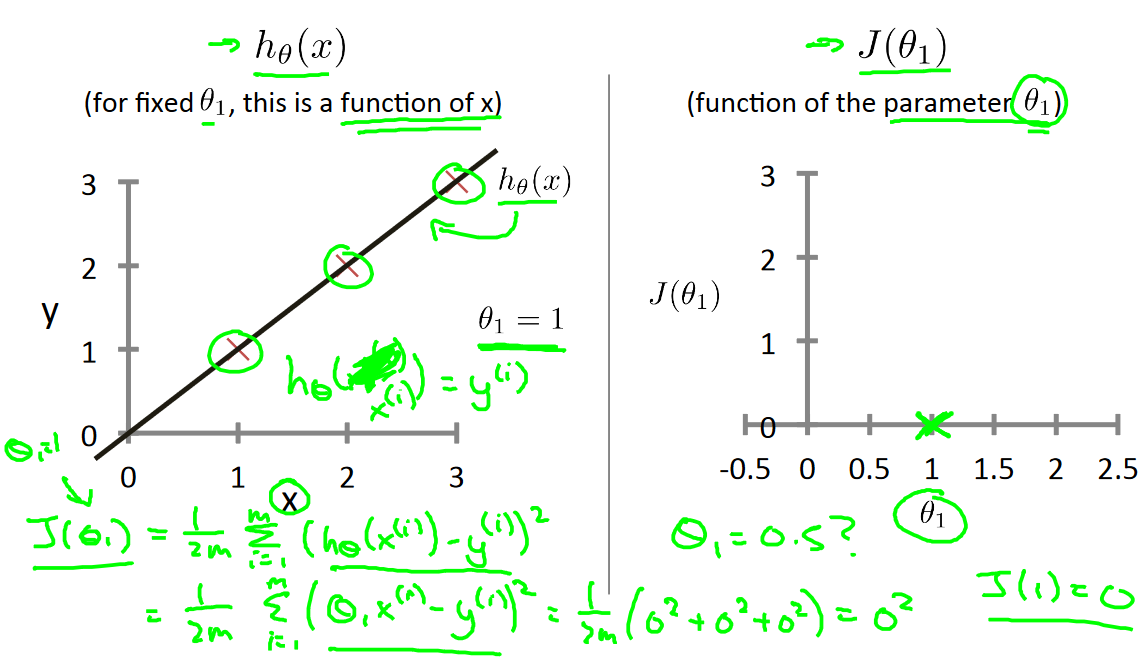
* We can measure the accuracy of our hypothesis function by using a **cost function**. This function is otherwise called the "Squared error function", or "Mean squared error".



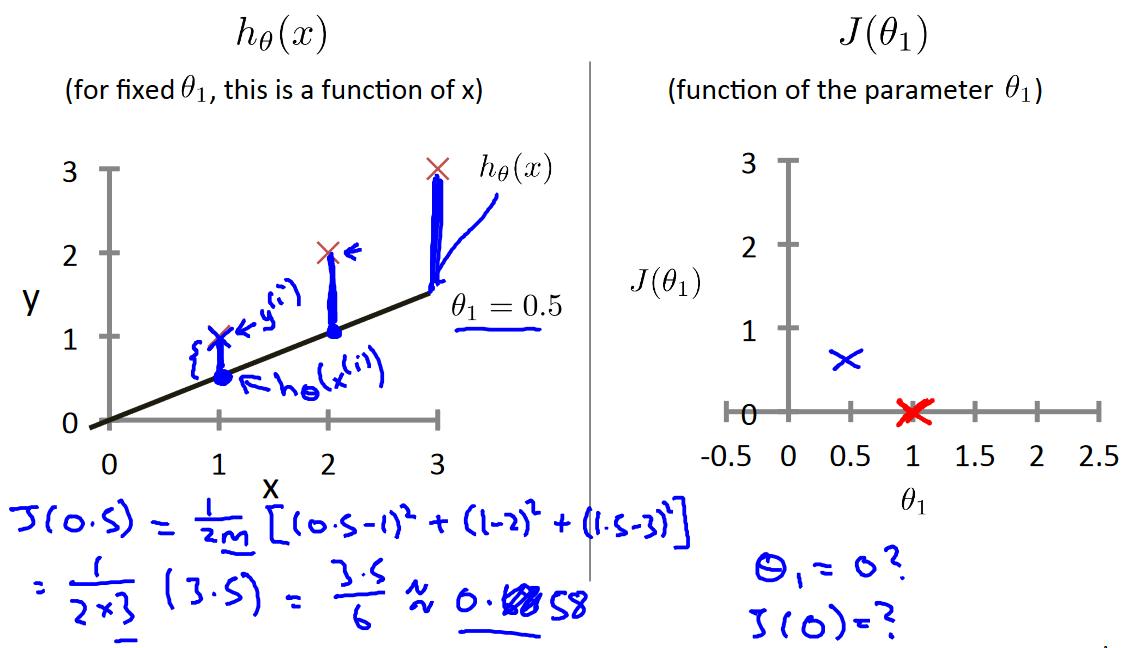
* This takes an average difference of all the results of the hypothesis with inputs from x's and the actual output y's.
* To break it apart, it is 12 x¯ where x¯ is the mean of the squares of hθ(xi)−yi, or the difference between the predicted value and the actual value.
* The mean is halved as a convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the 1/2 term.



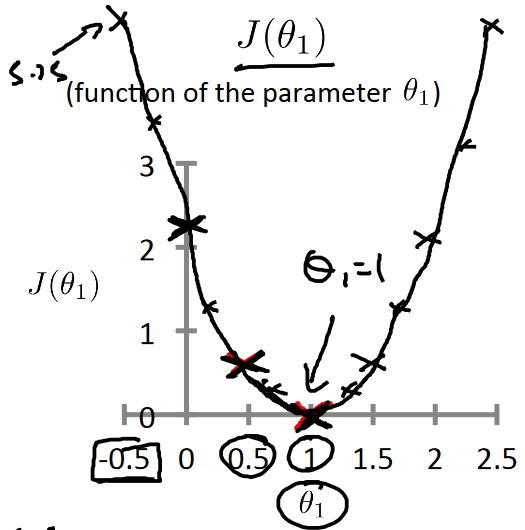
* Our training data set is scattered on the x-y plane. We are trying to make straight line (defined by hθ(x)) which passes through this scattered set of data. Our objective is to get the best possible line. The best possible line will be such so that the average squared vertical distances of the scattered points from the line will be the least. In the best case, the line should pass through all the points of our training data set. In such a case the value of J(θ0,θ1) will be 0:



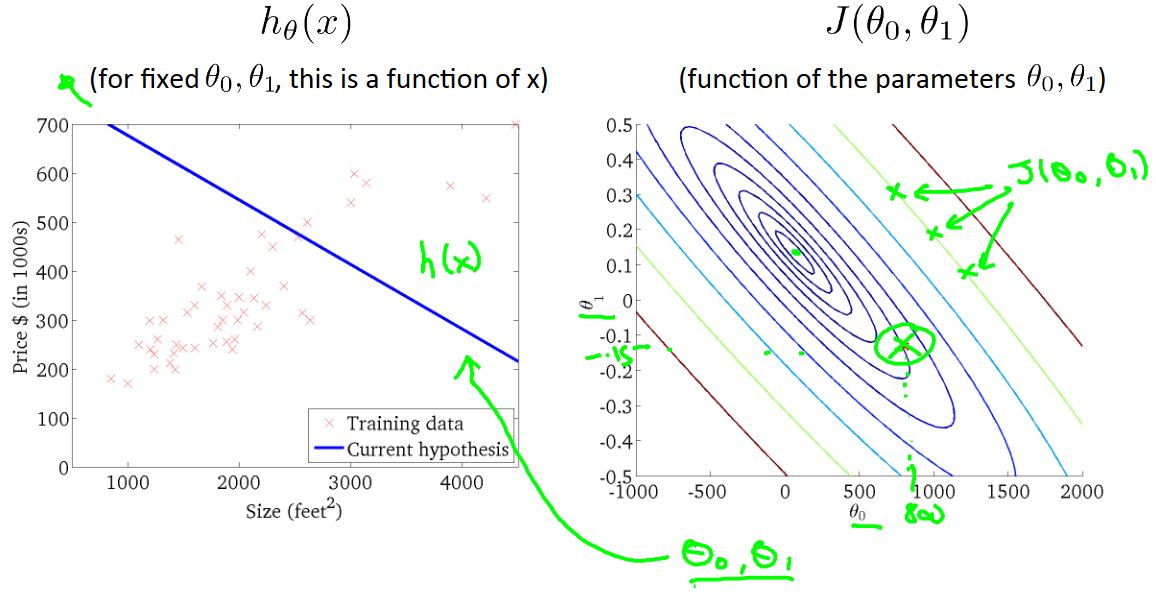
* When θ1=1, we get a slope of 1 which goes through every single data point in our model. Conversely, when θ1=0.5, we see the vertical distance from our fit to the data points increase. This increases our cost function to 0.58:



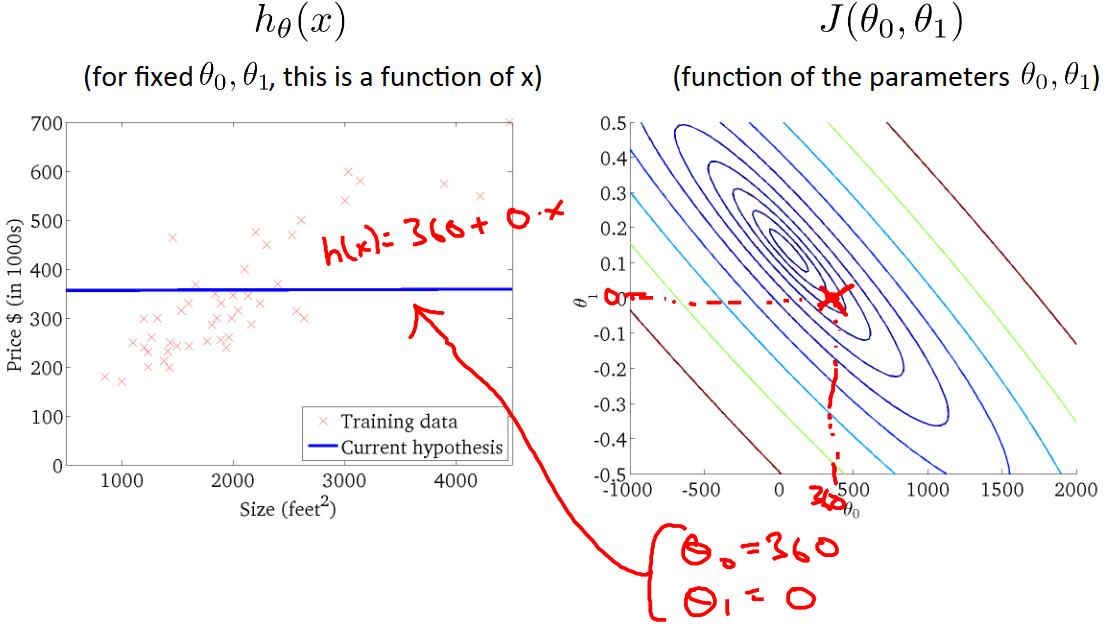
* Plotting several other points yields to the following graph (on the left); for J of two parameters θ0 and θ1 – it would be like plot on the right:

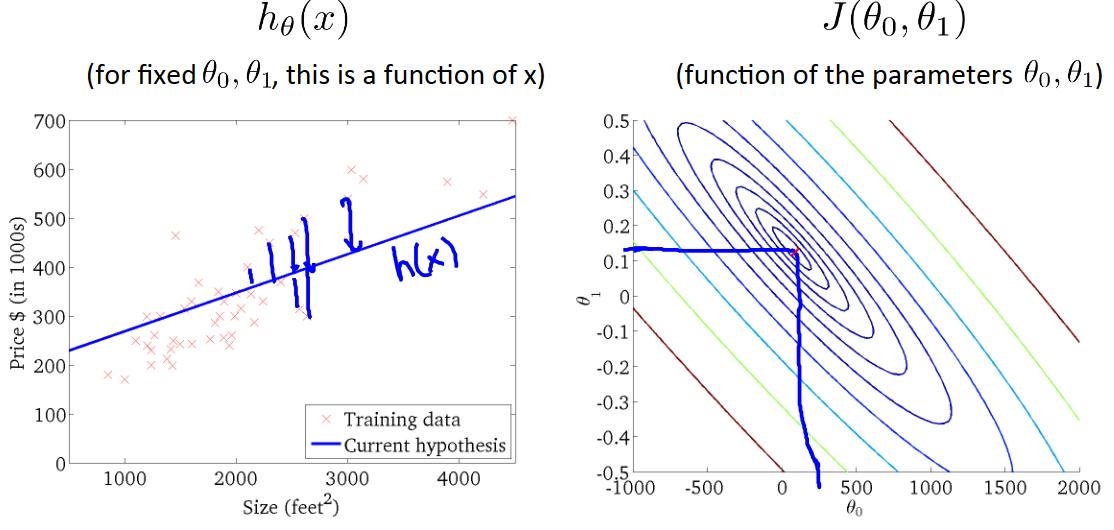
* Thus as a goal, we should try to minimize the cost function. In this case, θ1=1 is our global minimum.
* Cost function of two parameters:
  + Visualized with contour graph; a contour line of a two variable function has a constant value at all points of the same line. Taking any color and going along the 'circle', one would expect to get the same value of the cost function (it is on axis z). For example, the three green points found on the green line above have the same value for J(θ0,θ1) and as a result, they are found along the same line. The circled x displays the value of the cost function for the graph on the left when θ0 = 800 and θ1= -0.15:



* + When θ0 = 360 and θ1 = 0, the value of J(θ0,θ1) in the contour plot gets closer to the center thus reducing the cost function error. Now giving our hypothesis function a slightly positive slope results in a better fit of the data:

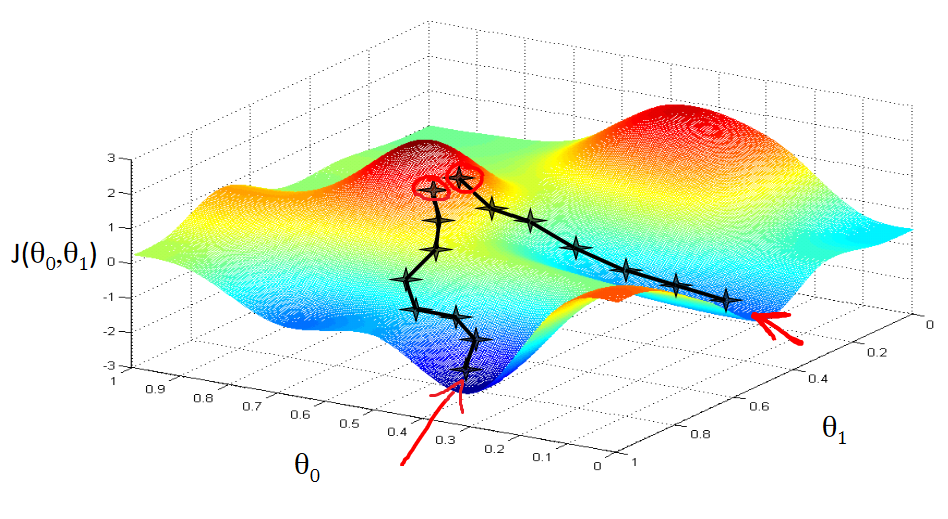


* + The graph below minimizes the cost function as much as possible and consequently, the result of θ1 and θ0 tend to be around 0.12 and 250 respectively. Plotting those values on our graph to the right seems to put our point in the center of the inner most 'circle'.



**Gradient descent:**

* On a graph – we put θ0 on the x axis and θ1 on the y axis, with the cost function on the vertical z axis. The points on our graph will be the result of the cost function using our hypothesis with those specific theta parameters.
* We will know that we have succeeded when our cost function is at the very bottom of the pits in our graph, i.e. when its value is the minimum. The red arrows show the minimum points in the graph. The graph shows two scenarios of gradient descent (depending on start point – one of the red-circled points).



* The way we do this is by taking the derivative (the tangential line to a function) of our cost function. The slope of the tangent is the derivative at that point and it will give us a direction to move towards. We make steps down the cost function in the direction with the steepest descent – it is determined by the partial derivative of J(θ0,θ1). The size of each step is determined by the parameter α, which is called the learning rate (a smaller α would result in a smaller step and a larger α results in a larger step).
* The gradient descent algorithm is (repeat until convergence):



where j=0,1 represents the feature index number.

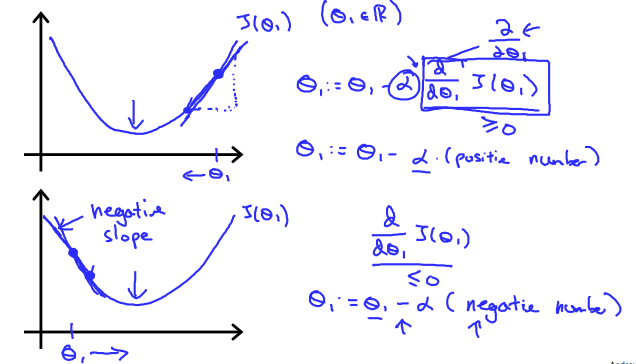
* At each iteration j, one should simultaneously update the parameters θ1,θ2,...,θn. Updating a specific parameter prior to calculating another one on the j(th) iteration would yield to a wrong implementation.



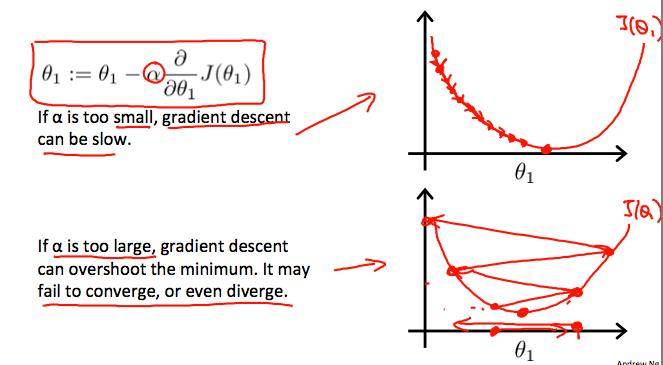
* Gradient descent for a single parameter θ1:



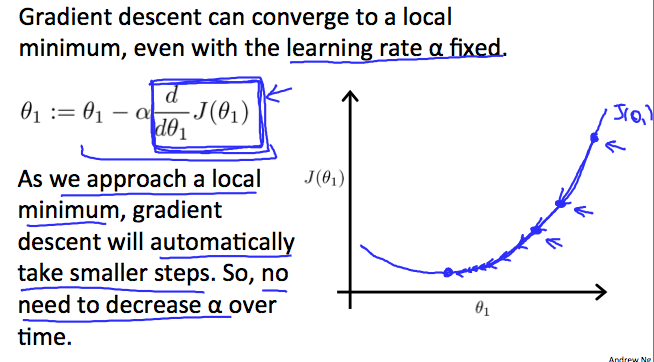
* + regardless of the slope's sign (for d/dθ1 \* J(θ1)) – when the slope is negative, the value of θ1 increases and when it is positive, the value of θ1 decreases (in a local minimum, this value is 0); θ1 eventually converges to its minimum value:



* + We should adjust our parameter α to ensure that the gradient descent algorithm converges and does that in a reasonable time:



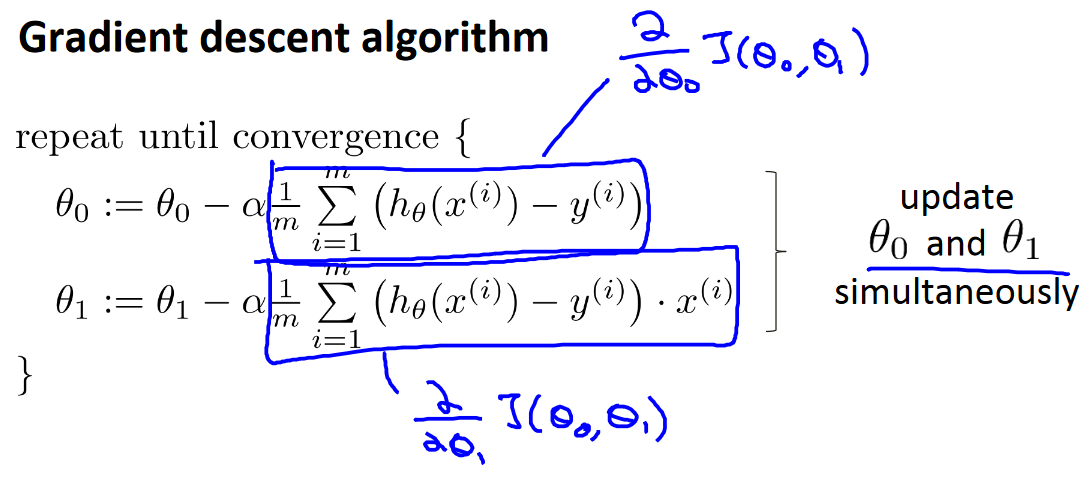
* + Gradient descent can converge even with a fixed step size α – because the derivative gets smaller while the slope gets less steepy:



* Gradient descent for linear regression:

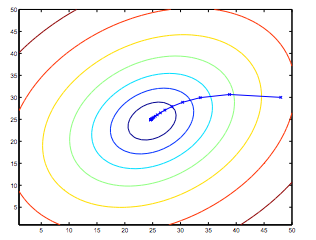


* + for j=0:
  + for j=1:



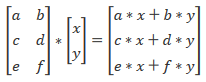
where m is the size of the training set, θ0 a constant that will be changing simultaneously with θ1 and xi, yi are values of the given training set (data).

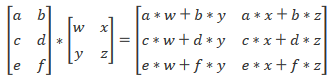
* + If we start with a guess for our hypothesis and then repeatedly apply these gradient descent equations, our hypothesis will become more and more accurate.
  + The ellipses shown above are the contours of a quadratic function. Also shown is the trajectory taken by gradient descent, which was initialized at (48,30). The x’s in the figure (joined by straight lines) mark the successive values of θ that gradient descent went through as it converged to its minimum:

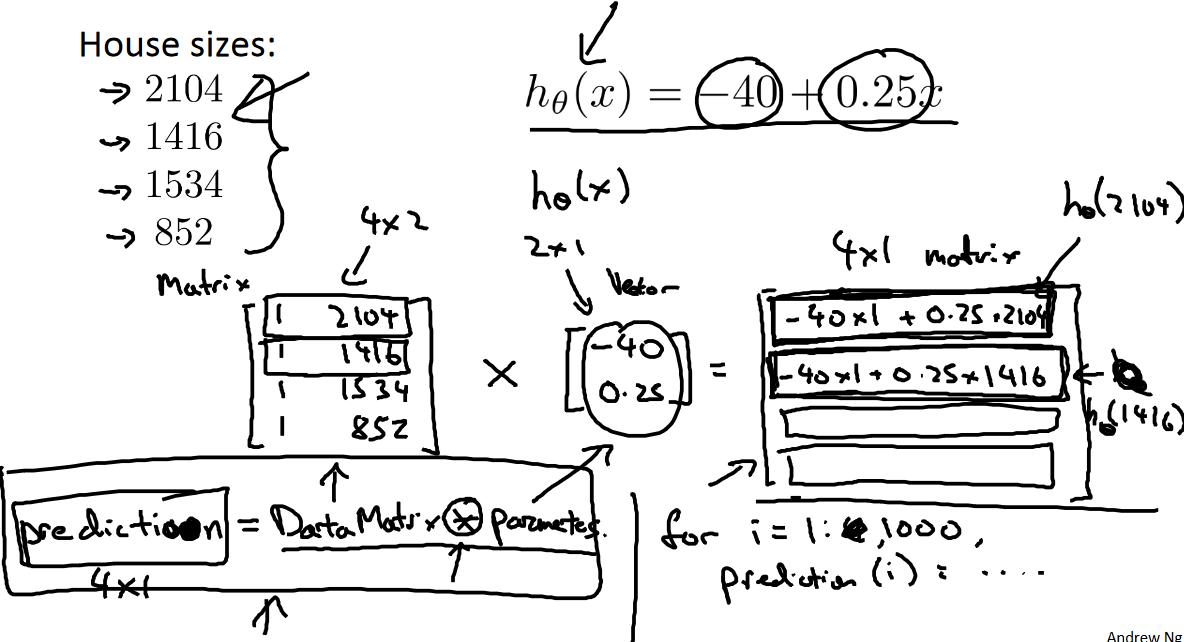


* + This method looks at every example in the entire training set on every step, and is called **batch gradient descent.**

**Matrixes and vectors using in ML:**



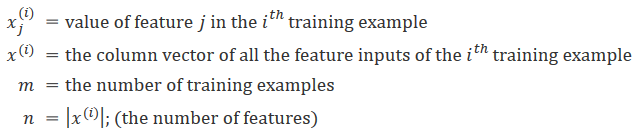


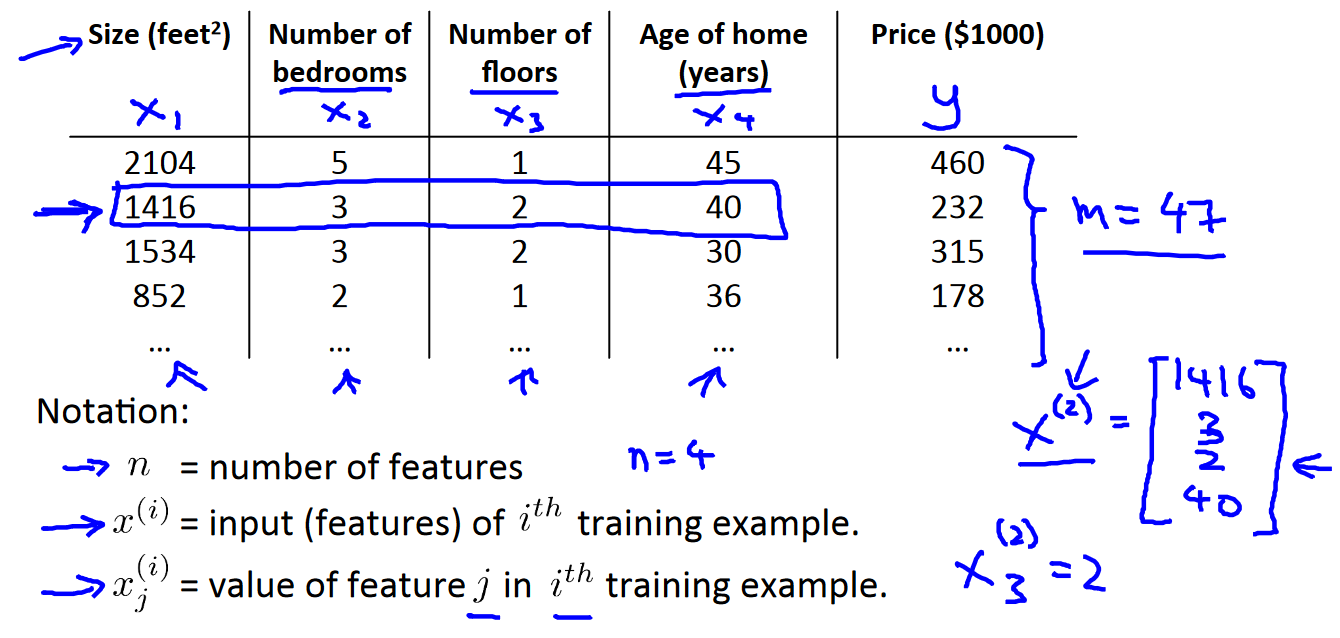




**Linear regresion with multiple variables:**

* notation for equations where we can have any number of input variables



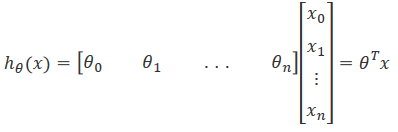


* multivariable form of the hypothesis function (accommodating multiple features):



where e.g.: θ0 is the basic price of a house, θ1 is the price per square meter, θ2 is the price per floor, etc.; x1 will be the number of square meters in the house, x2 the number of floors

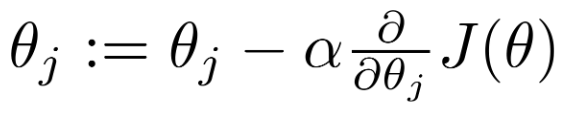
* vectorization of the hypothesis function for one training example (x0(i) = 1):

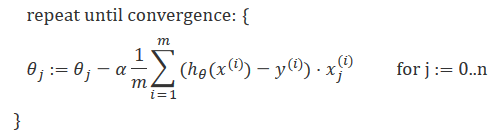


* the vectorised version of cost function:

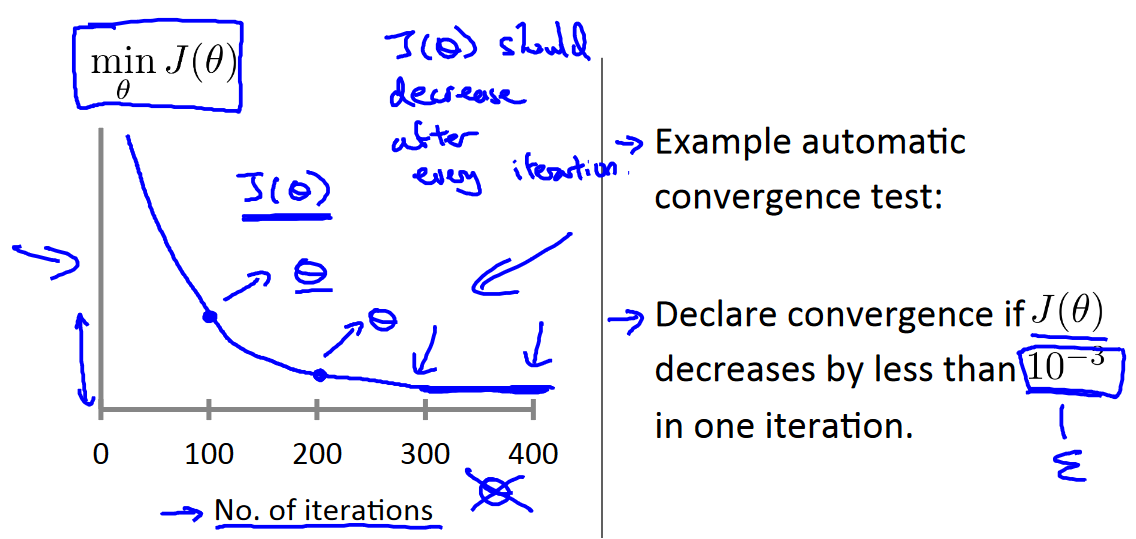


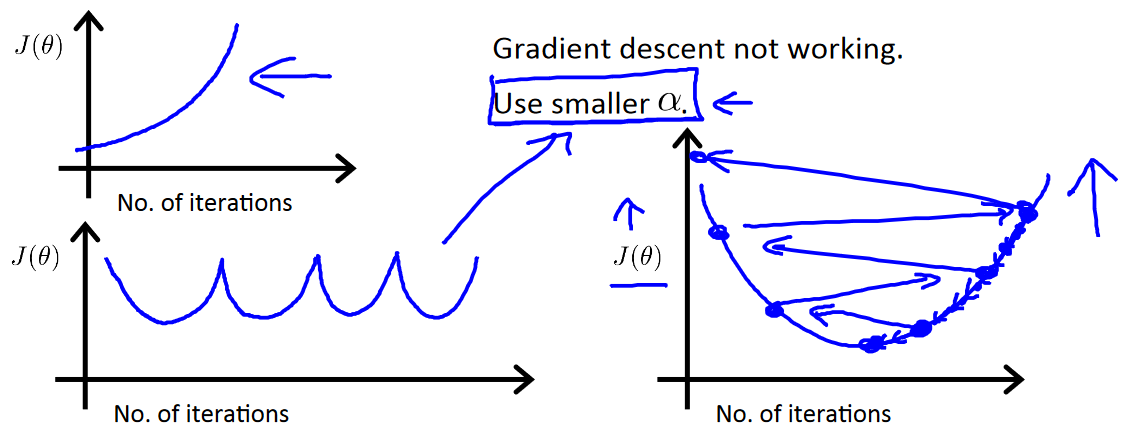
**Gradient descent for multiple variables:**





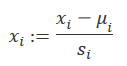
* Gradient descent debugging (how to make sure it works correctly):
  + make a plot with number of iterations on the x-axis and plot the cost function, J(θ) over the number of iterations of gradient descent
  + if J(θ) ever increases, then you probably need to decrease α
  + if learning rate α is sufficiently small, then J(θ) will decrease on every iteration
  + if α is too small: slow convergence
  + If α is too large: may not decrease on every iteration and thus may not converge; recommended of decreasing α by multiples of 3 (e.g. 0.003, 0.03, 0.3, 3 etc.)
* **Automatic convergence test – d**eclare convergence if J(θ) decreases by less than E in one iteration, where E is some small value such as 10−3. However in practice it's difficult to choose this threshold value.





**Feature normalization:**

* it’s modyfying ranges of our input variables so that they are all roughly the same range, idealny between -1 and 1, and usually from -0,5 to 0,5 (to speed up gradient descent)
* two techniques:
  + feature scaling – involves dividing the input values by the range (i.e. the maximum value minus the minimum value) of the inpur variable (x(i)/s(i))
  + mean normalization – involves substracting the average value for an input variable from the values for that input variable (x(i)-u(i))
* implementing both of above:



where μi is the **average** of all the values for feature (i) and si is the range of values (max - min), or si is the standard deviation

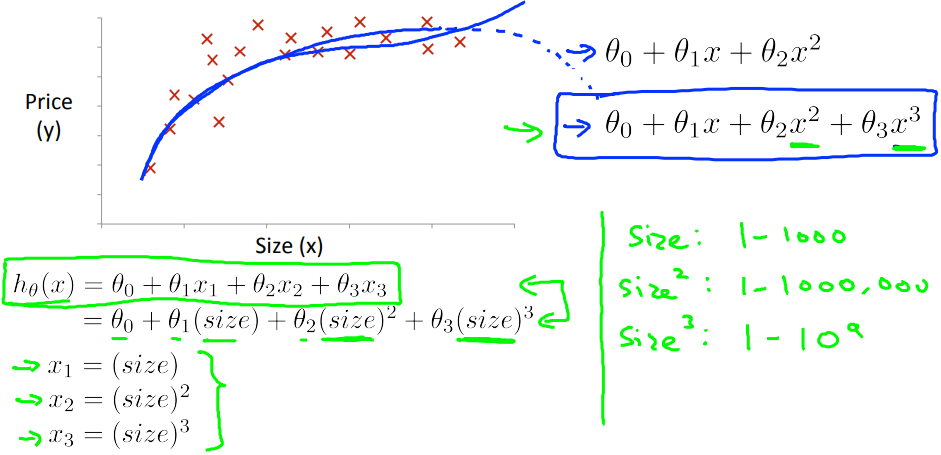
* example – xi is housing prices with range of 100 to 2000, with a mean value of 1000. Then:

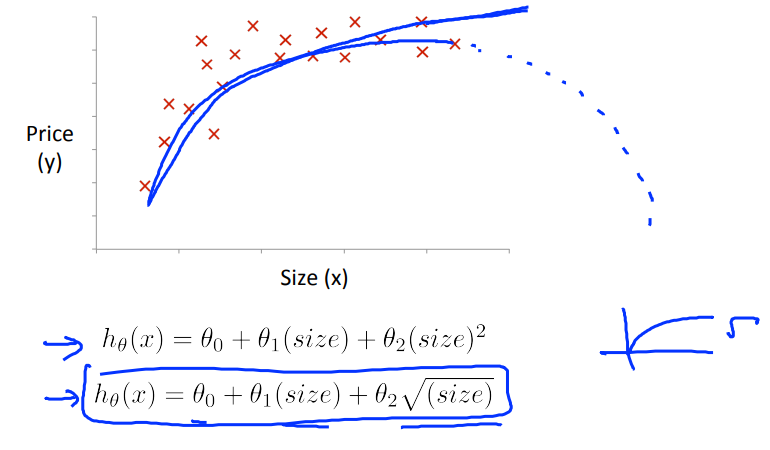


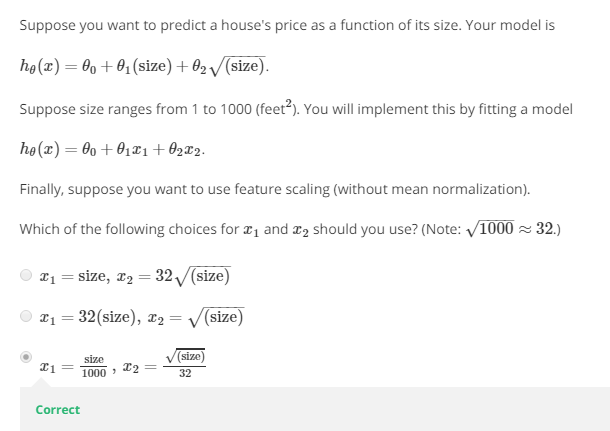
**Feature and polynomial regression:**

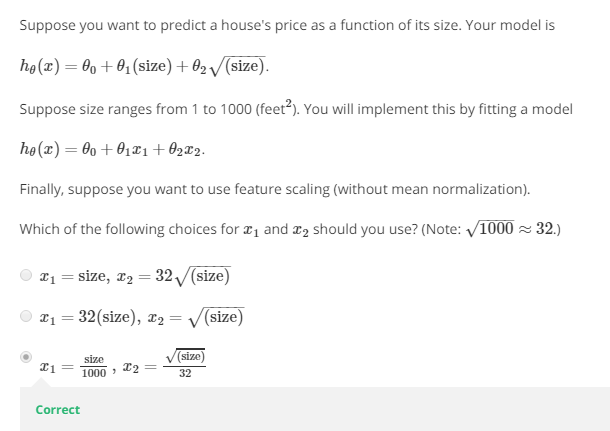
We can improve our features and the form of our hypothesis function in a couple different ways:

* by combining multiple features into one (e.g. x1 and x2 into a new feature x3 by taking x1\*x2, where x1 – frontage of a house, x2 – depth of a house)
* polynomial regression:
  + hypothesis function need not be linear (a straight line) if that does not fit the data well; we can **change the behavior or curve** of our hypothesis function by making it a quadratic, cubic or square root function (or any other form)
  + for example, if our hypothesis function is hθ(x)=θ0+θ1x1 then we can create additional features based on x1, to get the quadratic function hθ(x)=θ0+θ1x1+θ2x12 or the cubic function hθ(x)=θ0+θ1x1+θ2x12+θ3x13 (x2 = x12, x3 = x13), or a square root function: hθ(x)=θ0+θ1x1+θ2√x1

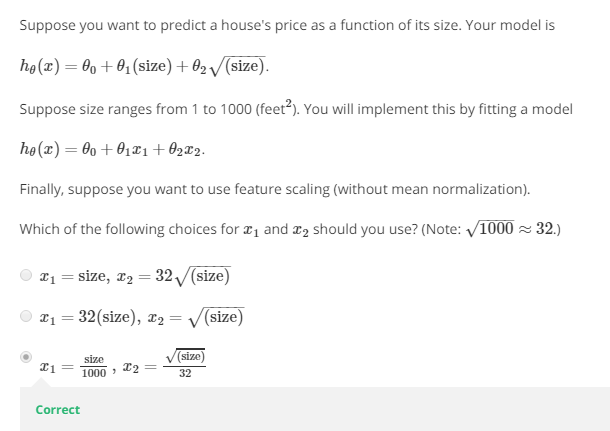




* + it’s very important to scale features used that way (because e.g. x1 has range 1-100, then x12 – 1-1000000 etc.), e.g. if you want to predict a house’s price as a function of its size, your model could be: ,

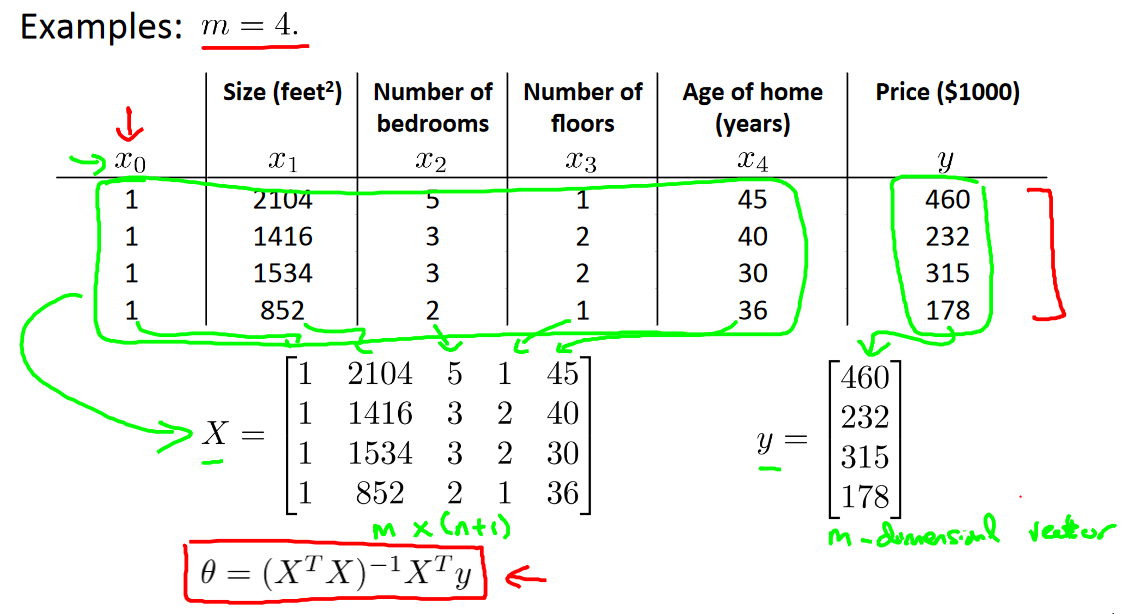
where size ranges from 1 to 1000 feet2 (so sqrt(size) ranges from 1 to 32). You will implement this by fitting a model: 

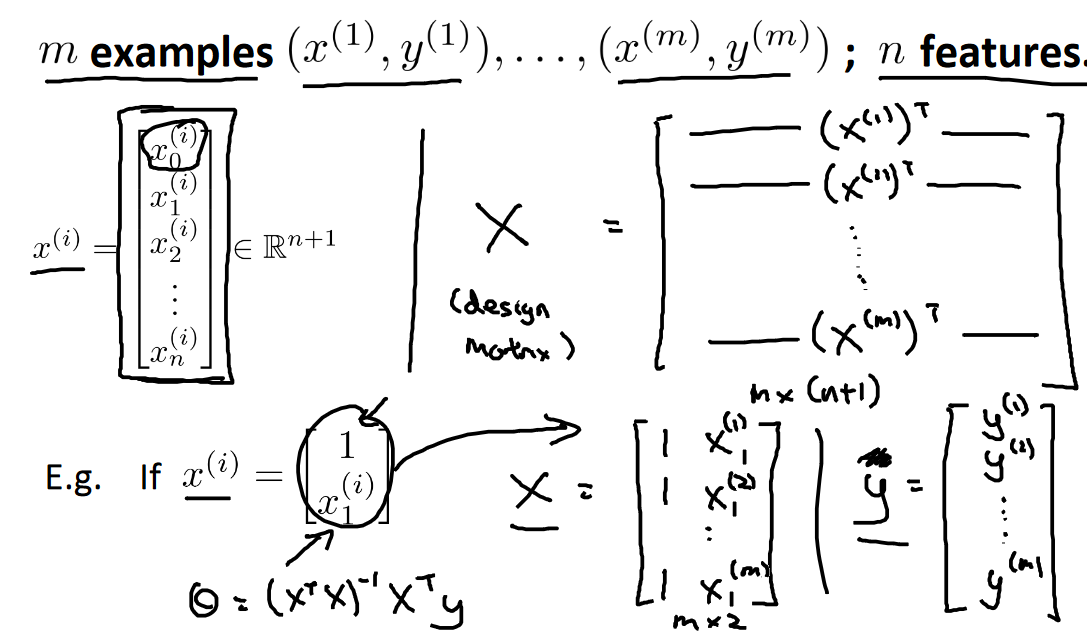
If you want to use feature scaling (without mean normalization), you will use:



**Normal equation:**

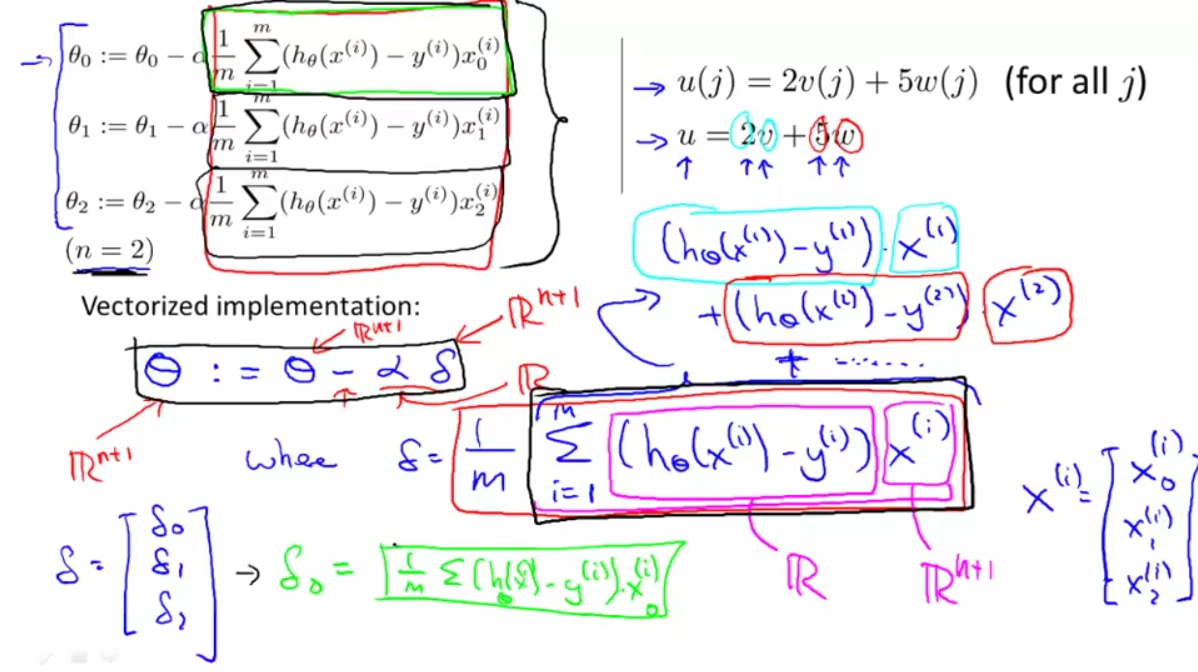
* a method of finding the optimum theta without iteration (we will minimize J by explicitly taking its derivatives with respect to the θj ’s, and setting them to zero)
* no need to do feature scaling and choosing alpha, but it has bigger computational complexity and works slower for large n (e.g. 10000) than gradient descent
* 





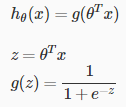
* in Octave: pinv(X’\*X)\*X’\*y

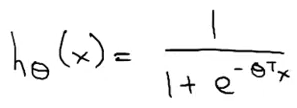
**Vectorization:**

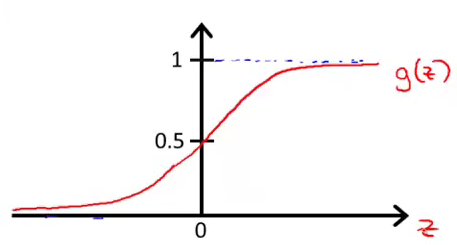


**Classification and Representation:**

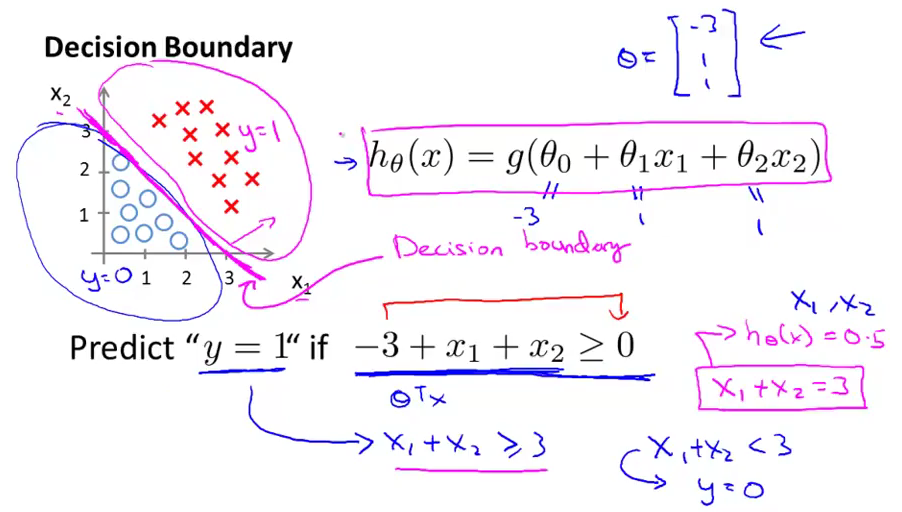
* in binary classification, y ∈ {0,1}, where 0 – negative class (e.g. benign tumor), 1 – positive class (e.g. malignant tumor)
* one approach is to predict 1 when h0(x) >= 0,5 and 0 when otherwise (0,5 – threshold)
* using linear regression as a classification algorithm, hθ(x) = θTx and it can be > 1 or < 0
* using logistic regression as a classification algorithm, 0 <= hθ(x) <= 1 and:

, where g – sigmoid/logistic function, so:



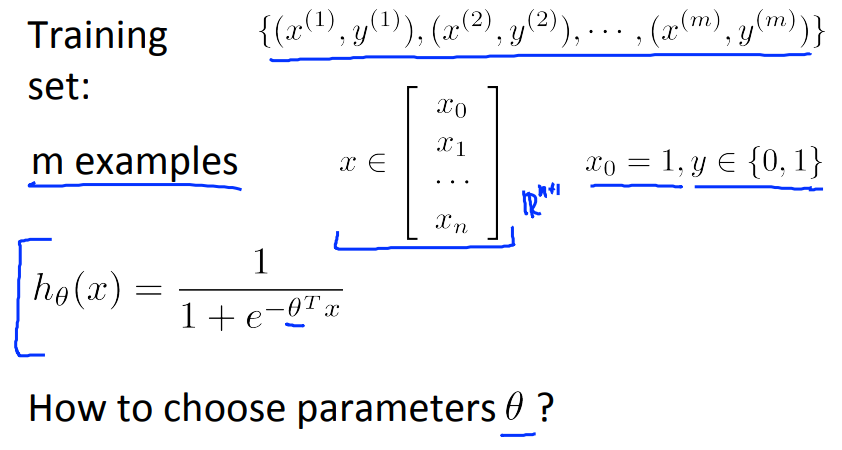


* function g(z) maps any real number to the (0, 1) interval, making it useful for transforming an arbitrary-valued function into a function better suited for classification
* using logistic regression, hθ(x) is interpreted as estimated probability that y = 1 on input x, parameterized by θ (so hθ(x) = P(y = 1 | x ; θ); e.g. x = [x0 x1] = [1 tumorSize and hθ(x) = 0.7, then there is 70% chance of tumor being malignant
* y = 1 when g(z) >= 0.5 (z >= 0), so hθ(x) >= 0.5 when θTx >= 0, e.g. (but it doesn’t need to be linear):



**Logistic Regression Model:**

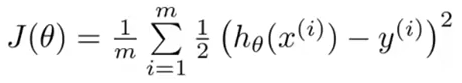
* how to choose parameters θ?



* cost function:



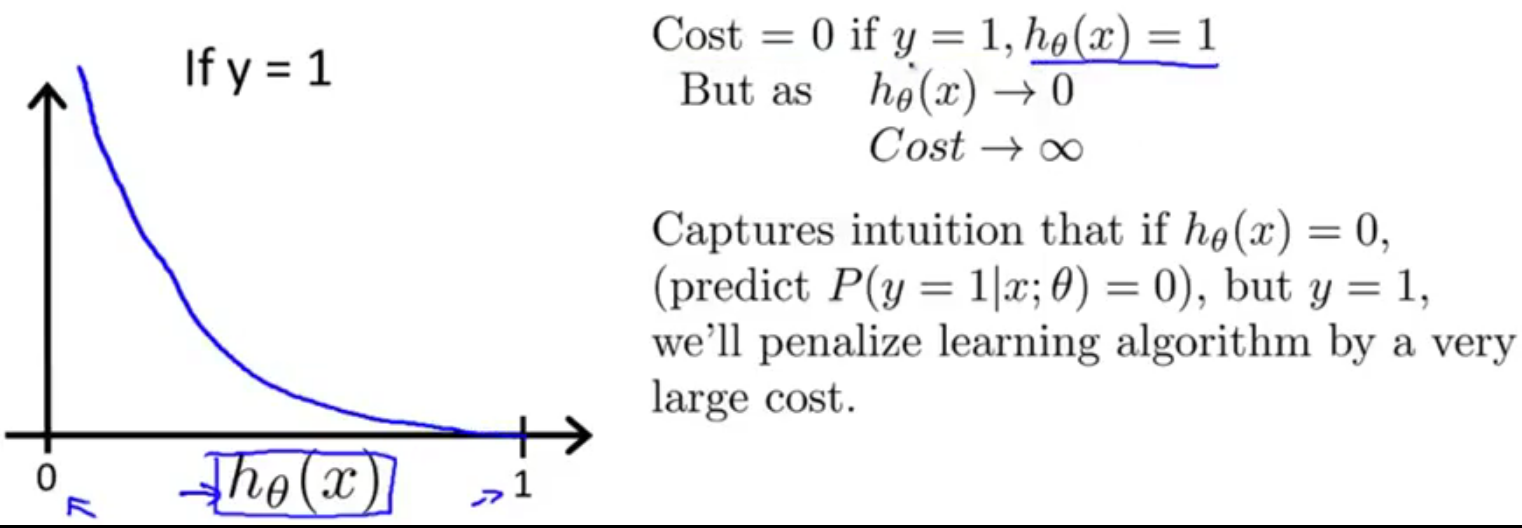
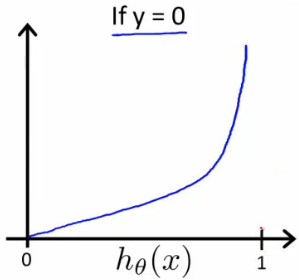
* we cannot use the same cost function that we use for linear regression:



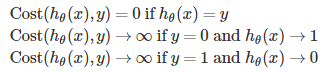
because the Logistic Function will cause the output to be wavy, causing many local optima (non-convex function, on which we can’t perform the gradient descent)

* in logistic regression, we use instead:



* the more our hypothesis is off from y, the larger the cost function output. If our hypothesis is equal to y, then our cost is 0:



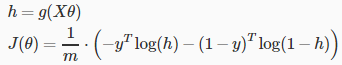
* simplified cost function for logistic regression:



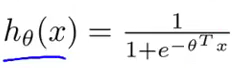
so:



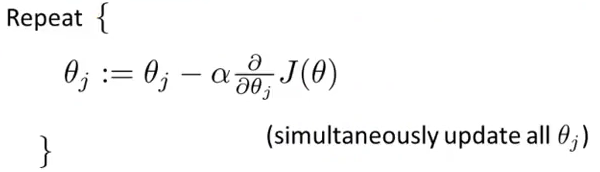
and vectorized implementation:



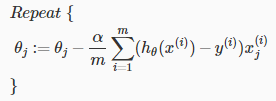
* then we have to fit parameters θ to minimize J(θ) and to make prediction given new x:



* we do that by gradient descent:



so:



so the algorithm is identical to linear regression, what is different is hθ(x) definition; vectorized:

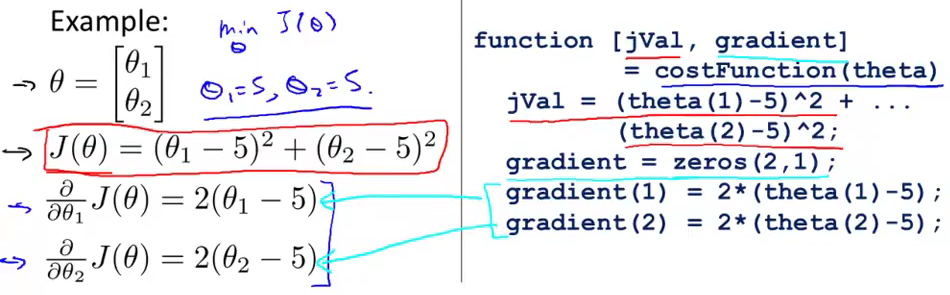


*θ* := *θ* – *α/m \** ​*XT* (*g*(*Xθ*) − *y*​)

* apart from gradient descent, there are a few others optimization algorithm for optimize cost function; they are more complex (there are libraries in Octave to implement them), but there’s no need to manually pick learning rate α and they can be faster:
  + conjugate gradient
  + BFGS
  + L-BFGS
* in order to use these, we need to provide a function that computes these two parameters (given input value θ):

, e.g.: , where:

jVal – cost function formula, gradient – partial derivatives for each θ; e.g.:



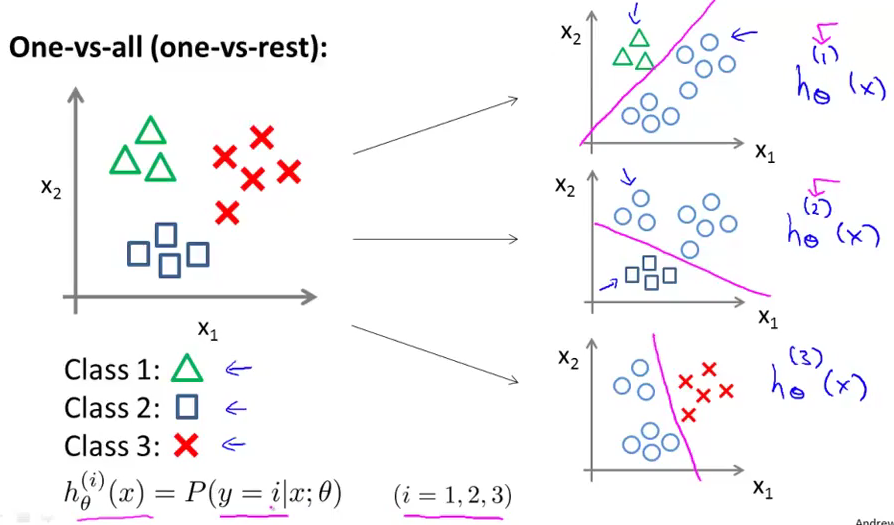
and then we can call the advanced optimization function “fminunc()” with the “optimset()” function:



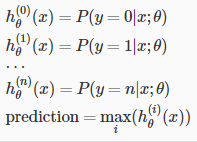
“optimset()” creates an object containing the options we want to send to “fminunc()” – here: providing gradient is on and max number of iteration is 100; @costFunction is the pointer to the previous function

**Multiclass classification:**

* used when we have more than 2 classes (y = {0,1...n}, but classes are indexed from 1), e.g. y = 1 patient not ill, y = 2 – cold, y = 3 – flu
* we turn our training set to 3 binary classification problems; in which one there is one class as positive and the rest as negative example:

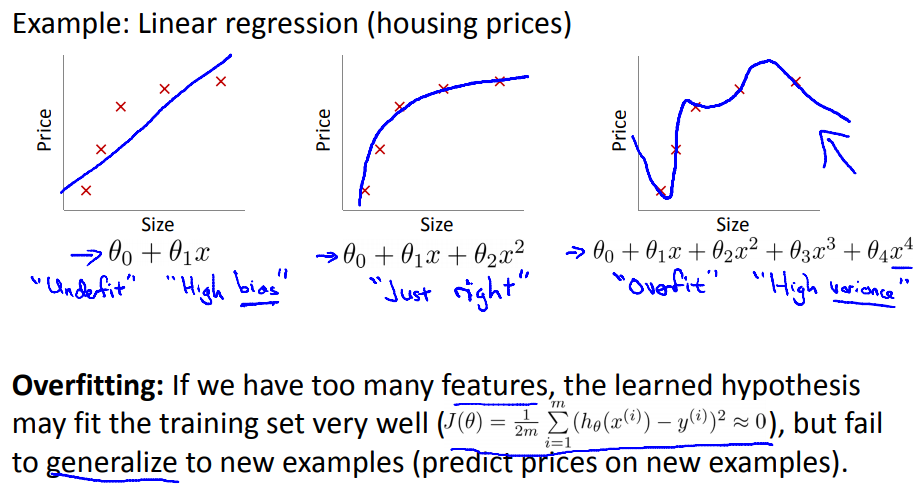


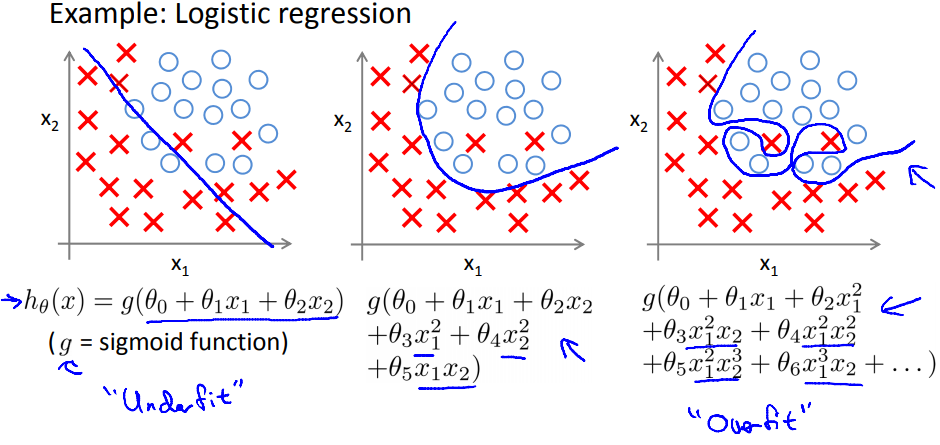
* so we train a logistic regression classifier *h*θ(*x*) for each class to predict the probability that y = i
* making prediction on a new x – pick the class that maximizes *h*θ​(*x*):



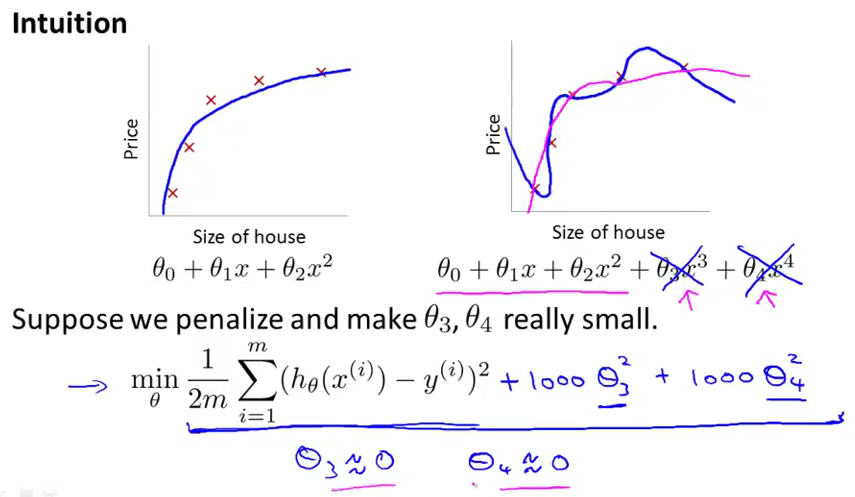
**The problem of overfitting:**

* examples of overfitting:



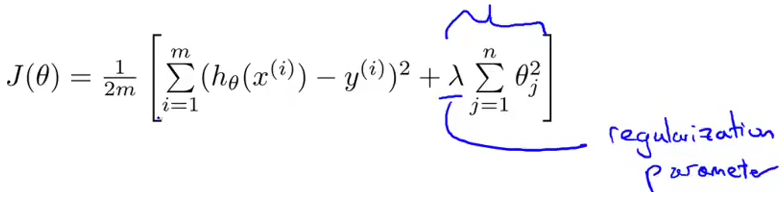


* overfitting may become a problem when we have a lot of features and very little training data
* options to address overfitting:
  + reduce number of features – manually select more important features to keep, or automatically (using model selection algorithm)
  + regularization – keep all the features, but reduce magnitude/values of parameters θj; works well when we have a lot of features, each of contributes a bit to predicting y
* cost function – we can “penalize” some of the features by reducing their weights, to eliminate their influence without getting rid of them, e.g. we add to the cost function formula:

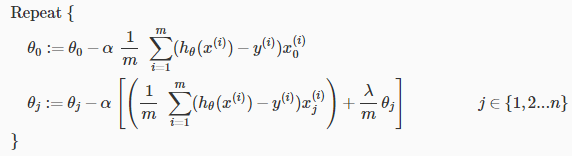


and set θ3 and θ4 to the values near to zero

* smaller values for parameters θ will make hypothesis simpler and less prone to overfitting
* if there is a lot of parameters, the best option is to reduce values for all of the thetas (beside θ0) by adding regularization term:



* lambda (the regularization parameter) determines how much the costs of our theta parameters are inflated; controls a trade off between two different goals (fitting well to the training data and keep the parameters small to keep hypothesis small); if it will be to large, e.g. 1010, it can cause even underfitting (all thetas near 0, horizontal line)
* regularization can be applied both to linear and logistic regression
* regularization in linear regression:
  + gradient descent:

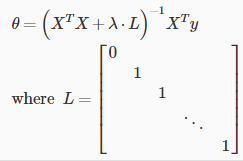


or:



where  will be always < 1 (reducing the value of θj by some amount on every update; while the second part of the formula above is exactly the same as original gradient descent)

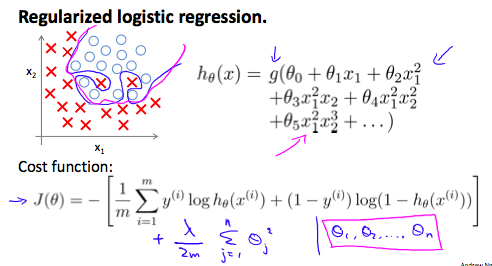
* + normal equation – the same equation, except we add another term inside the parentheses:



where L is a matrix with 0 at the top left and 1's down the diagonal, with 0's everywhere else, with dimension (n+1)×(n+1);

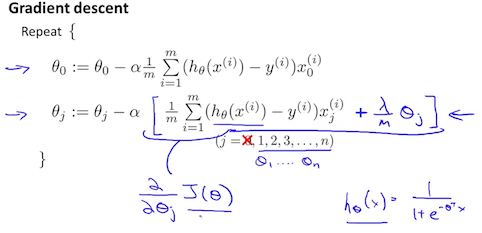
if m < n, then XTX is non-invertible, but when we add the term λ⋅L, then XTX + λ⋅L becomes invertible.

* regularization in logistic regression:
  + cost function – we just add to the formula:



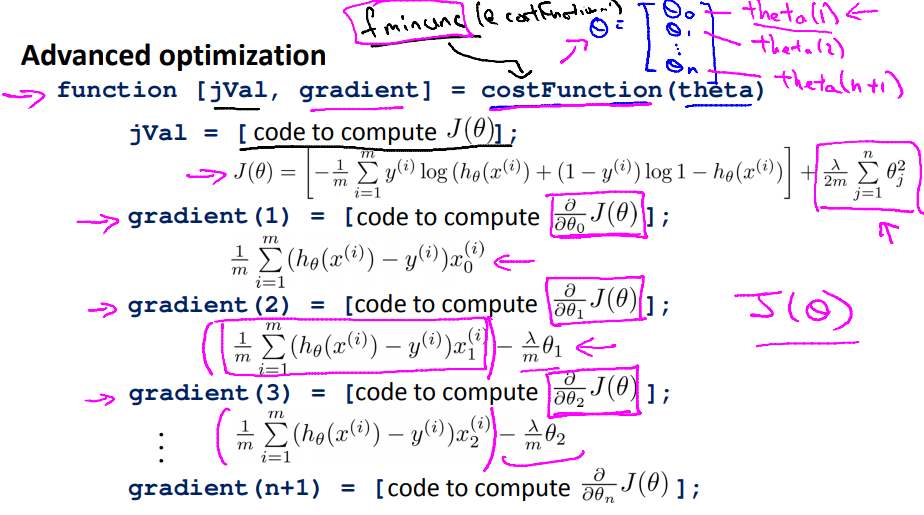
where pink line is regularized function and blue line – not regularized (overfitted)

* + gradient descent:



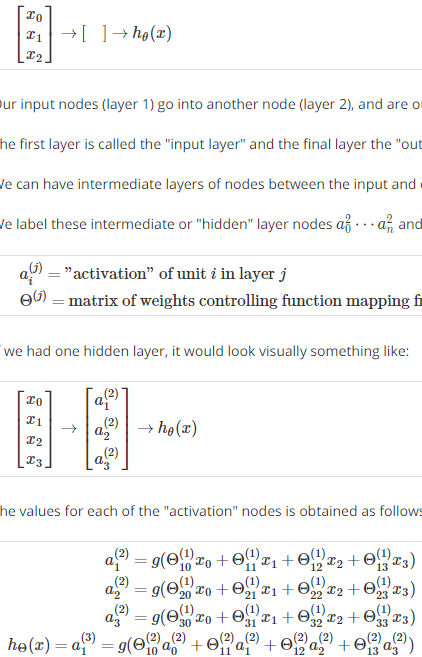
so the formula is the same as for linear regression, except hθ(x) is different

* regularization in advanced optimization methods:

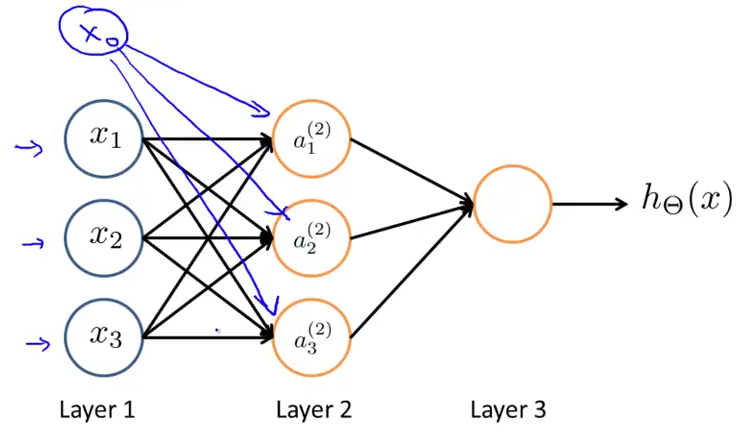


**Neural networks:**

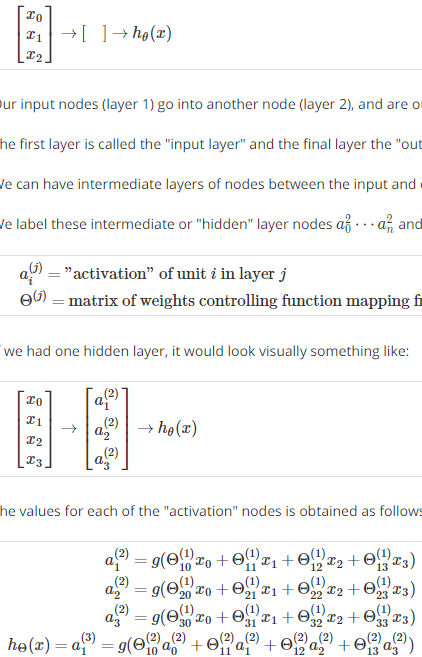
* for many features, for non-linear classification better is to use neural networks than logistic regression (in which e.g. n = 100, we have app. 5000 features (with all quadratic terms) in our hypothesis – O(n2/2); all cubic terms – O(n3); example: let our training set be a collection of 50 x 50 pixel black-and-white photographs, and our goal will be to classify which ones are photos of cars. Our feature set size is then n = 2500 pixels intensities (7500 if RGB) if we compare every pair of pixels; so our total features will be about 25002 / 2 = 3125000, which is very impractical).
* neuron model – each neuron has inputs (x1, x2, …), performs some calculations with it and has outputs (hθ):



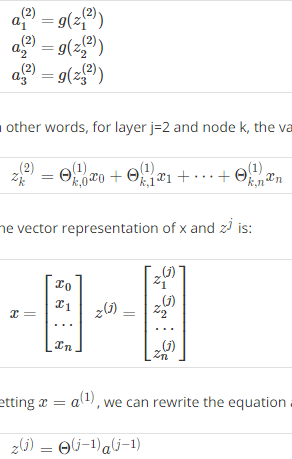
* θs are called weights and x0 = 1 is called bias. Sigmoid (logistic) function is called an activation function
* artificial neural network:



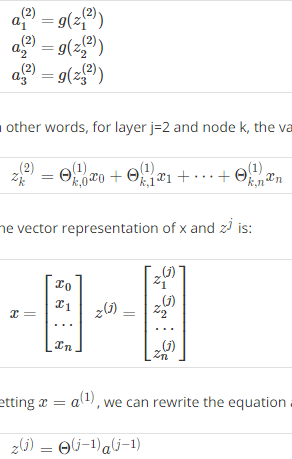
* layer 1 is an input layer, layer 3 – output layer and intermediate layers – hidden layers, and their neuron (called activation units) are denoted such as ai(j) where i – neuron number, j – layer number
* θ(j) – matrix of weights controlling function mapping from layer j to layer j + 1
* so in this example:



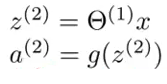
* here, θ(1) is 3 x 4 matrix – there is a rule, that if network has n units in layer j and m units in layer j + 1, then θ(j) will be of dimension m x (n + 1), where this + 1 is for bias
* forward propagation – vectorized implementation:
  + a new variable zk(j)​ that encompasses the parameters inside our g function:



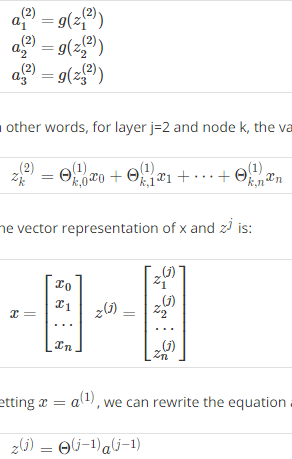
* + so we have vectors:



* + and we can calculate:



where a(2) and z(2) are both in this case 3-dimensional vectors; x can be written as a(1), then:



* + while calculating next layers (here: layer 3 – output layer), we should also add a0(2) = 1 as a bias