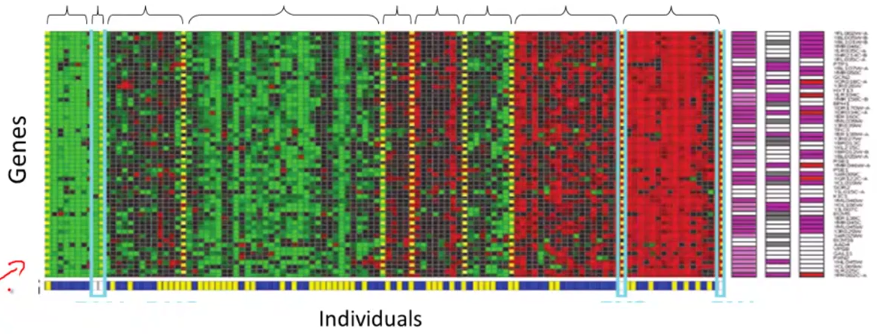
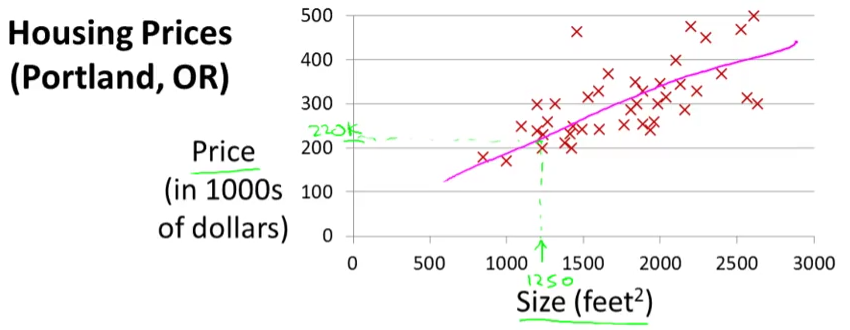
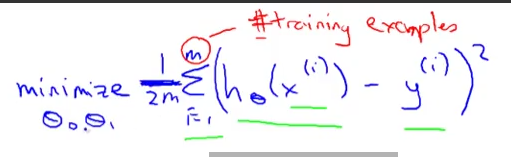
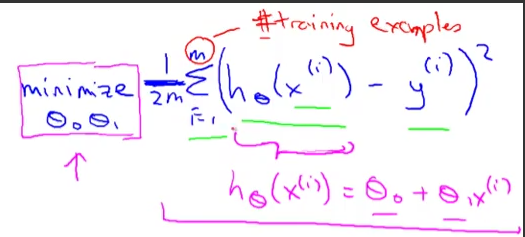
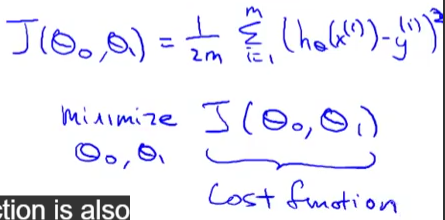
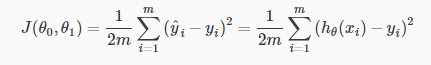
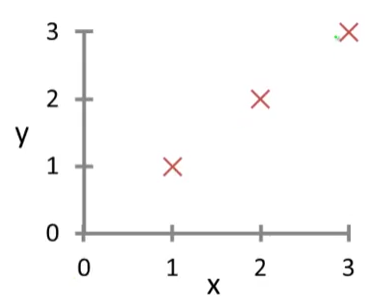
* **Machine Learning** - Grew out of work in AI + is a new capability for CPUs
* Database mining
* Large datasets from growth of automation/web (Web click data, medical records, biology, engineering, etc.)
* Applications that can’t be programmed by hand. (Autonomous helicopter, handwriting recognition, most of NLP, Computer Vision
* Self-customizing programs (Amazon/Netflix product recommendations, Understanding human learning (brain, real AI))
* 1959 Definition of **Machine Learning** = Field of study that gives CPUs the ability to learn w/out being explicitly programmed.
* 1998 Definition of **Machine Learning** = A program is said to learn from experience E w/ respect to some task T and some performance measure P, if its performance on T, as measured by P, improves w/ experience E
* **Supervised Learning 🡪** given a data set and *already know* what our correct output (“right answer”) should look like, while also having the idea that there is a relationship between the input and the output.
* Supervised learning problems are categorized into **regression** and **classification** problems.
* regression 🡪 trying to predict results within a *continuous* output, or we are trying to *map input variables to some continuous function*.
* Classification 🡪 trying to predict results in a *discrete* output (0 or 1), or we are trying to map *input variables into discrete categories*.
* Example 1: Given data about the size of houses on the real estate market, try to predict their price 🡪 Price = a function of size 🡪 a continuous output, so this is a regression problem.
* We could turn this example into a classification problem by instead making our output = *whether a house sells for more than or less than the asking price.*
* Here we are classifying the houses based on price into 2 discrete categories (> or <)
* Example 2:
* (a) Regression - Given a picture of a person, predict age based on the given picture
* (b) Classification - Given a patient w/ a tumor, predict whether it is malignant or benign.
* **Unsupervised learning** allows us to approach problems w/ little/no idea what results should look like 🡪 given a data set w/ no labels or what each it, find some structure w/in it
* can derive structure from data where we don't necessarily know the effect of the variables.
* can derive this structure by **clustering** data based on relationships among variables in the data.
* NO feedback based on the prediction results.
* Example 1: Clustering:
* Take a collection of 1,000,000 different genes and find a way to automatically group them into *groups* that are *somehow similar*/related by different variables, such as lifespan, location, roles, and so on.
* Take a collection of genes being expressed in individuals (via color gradient) and group individuals into similar groups without knowing types of people in advance (no right answer)

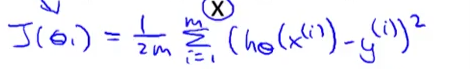
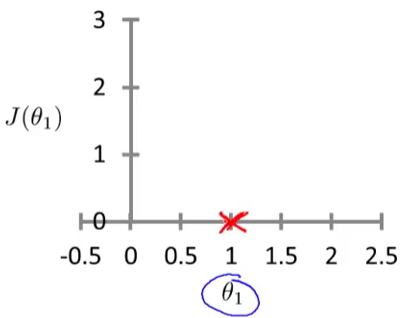
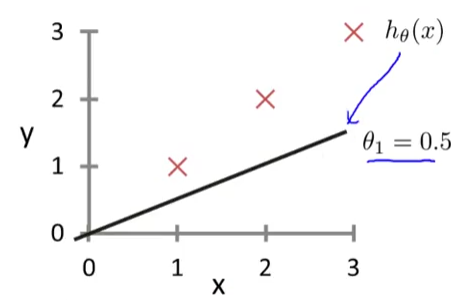
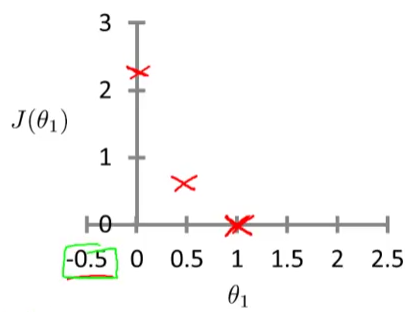
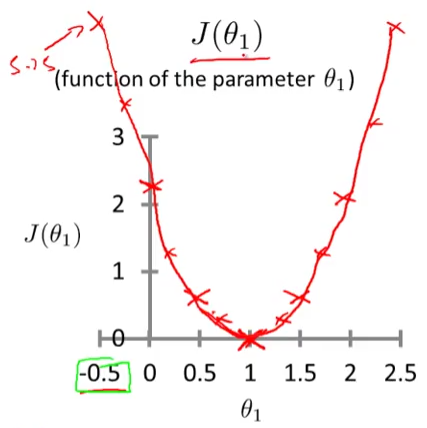
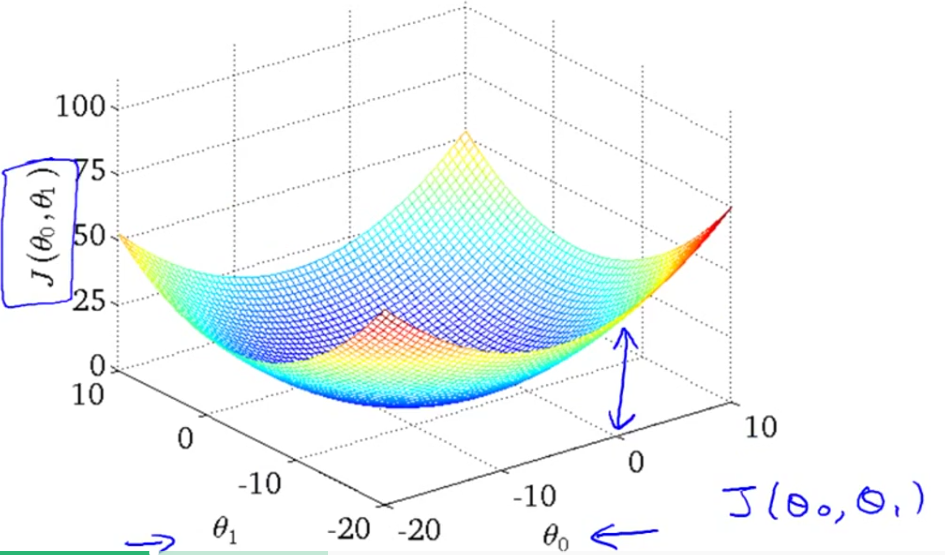
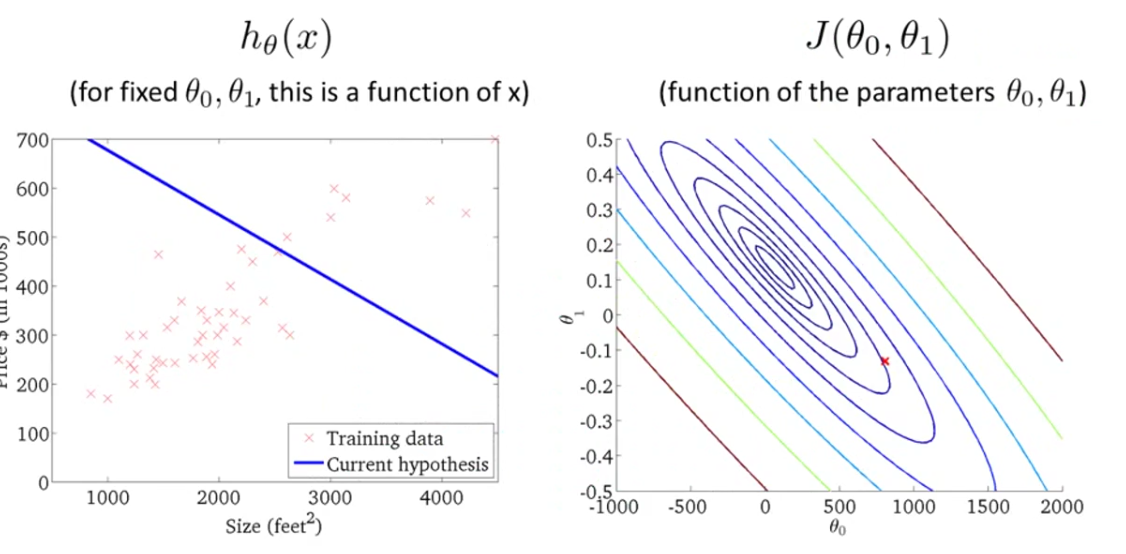
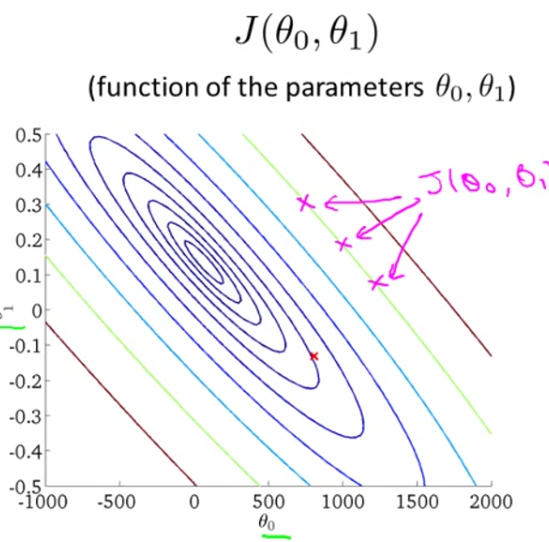
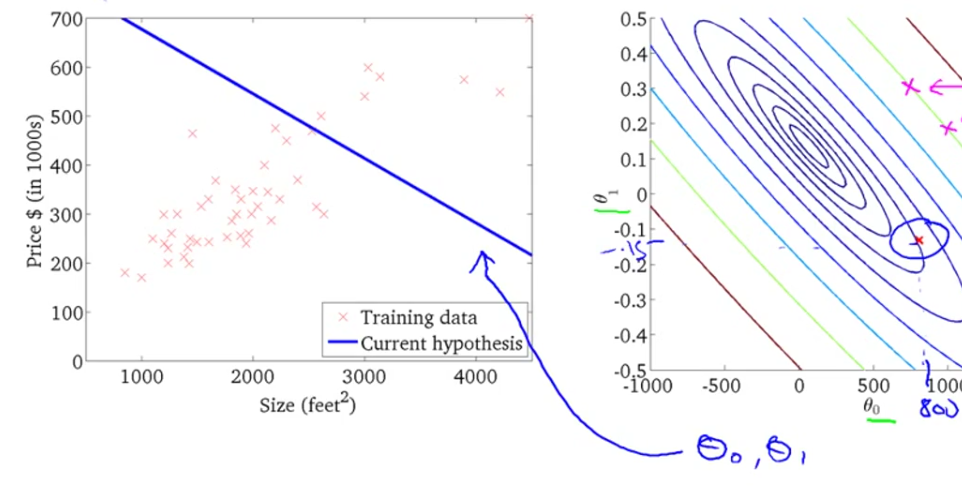
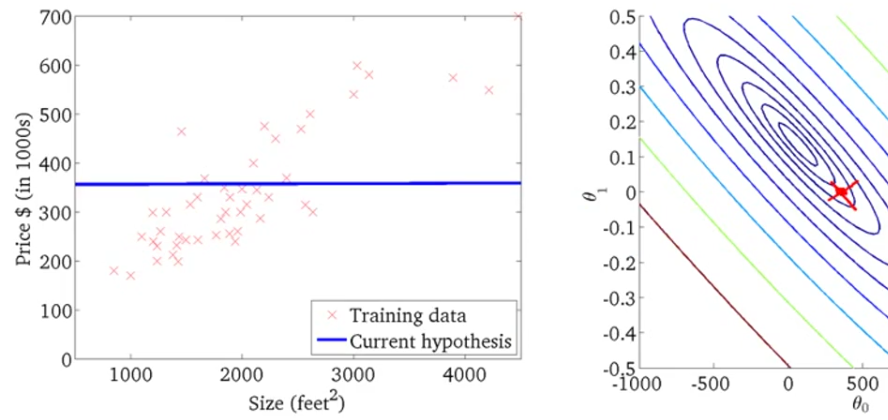
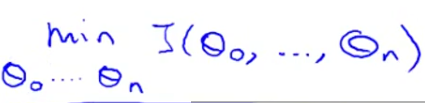
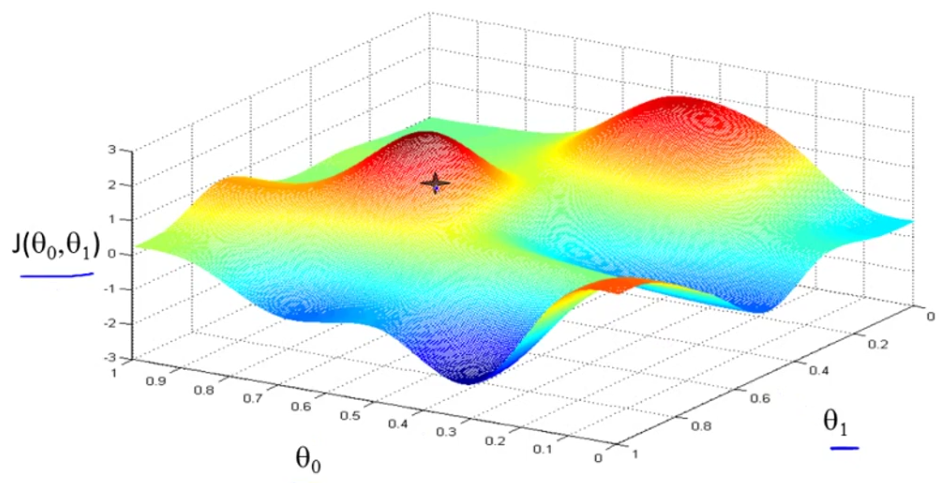
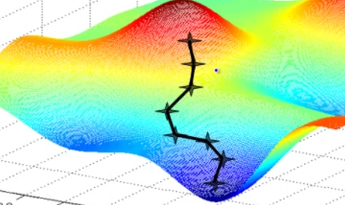
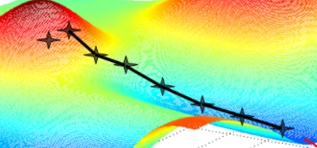


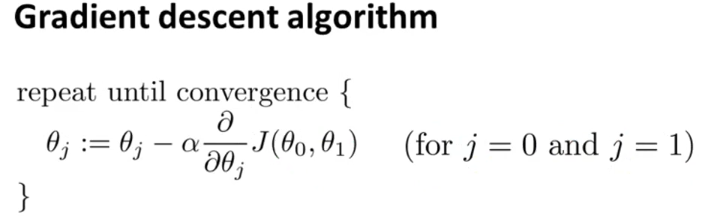
* Google News 🡪 looks throughout stories on the web and groups them into cohesive news stories (multiple URL’s in each cluster to display together)
* Organize large CPU clusters to see which machines work better together
* Social Network analysis 🡪 by knowing who you email/Facebook friends, etc. can we automatically ID cohesive groups of friends that all know each other
* Market Segmentation 🡪 look at customer data set and group customers into different market segments to more efficiently market to different segments
* Astronomical Data Analysis 🡪 useful theories on how galaxies are formed
* Example 1: Non-clustering:
* 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 [cocktail party](https://en.wikipedia.org/wiki/Cocktail_party_effect)).
* Imagine a party in a room full of people all sitting around + talking at the same time = all these overlapping voices b/c everyone is talking at the same time = almost hard to hear the person in front of you.
* At a cocktail party w/ 2 people talking at the same time w/ 2 microphones in the room at 2 different distances 🡪 each mic records a different combination of these 2 voices.
* Maybe speaker 1 is a little louder in mic 1 and speaker 2 is a little bit louder on mic 2, b/c the 2 microphones are at different positions relative to the 2 speakers, but each mic would cause an overlapping combination of both voices.
* We can take these 2 mic recorders and give them to an Unsupervised Learning algorithm called the **cocktail party algorithm**, and tell it to find structure in this data for us.
* The algorithm will “listen“ to these audio recordings and “it sounds like the 2 recordings are being added together or have been summed together to produce these recordings”.
* Moreover, the cocktail party algorithm will *separate out* these 2 audio sources that were being added/summed together and form other recordings on its own
* Can separate out 2 people speaking different languages into 2 separate recordings of each language, or separate out 1 person speaking and background music into 2 separate recordings
* It might seem complicated to implement this +, say, do a lot of coding to do audio processing or link into like a bunch of synthesizer Java libraries that process audio,
* It turns out the algorithm to do this can be done w/ 1 line of code, but it took researchers a LONG time to come up w/ this 1 line of code.
* **[W,s,v] = svd((repmat(sum(x.\*x,1),size(x,1),1).\*x)\*x’);**
* This is not an easy problem, but it turns out that when you use the right programming environment, many learning algorithms can be a really short program
* We will use the **Octave programming environment,** an open-source software (or MatLab)
* It turns out in Silicon Valley, for a lot of ML algorithms, what we do is 1st prototype software in Octave, b/c software in Octave makes it incredibly fast to implement learning algorithms.
* Above, the **SVD** function (**singular value decomposition**) turns out to be a linear algebra routine that is just built into Octave.
* If trying to do this in C++ or Java, it would be many lines of code, linking complex C++ or Java libraries.
* Most learn much faster if using Octave as a programming environment and learning and prototyping tool, as it'll let one learn and prototype learning algorithms much more quickly.
* Many people in Silicon Valley companies in fact use an algorithm like Octave to first prototype a ML algorithm, + only after getting it to work, then migrate it to C++ or Java, etc.
* By doing things this way, you can often get an algorithm to work much faster than if starting in C++.
* So **Unsupervised Learning =** a learning setting where you give an algorithm a ton of data and just ask it to find structure in the data for us.
* use a clustering algorithm to cluster articles together about the same story
* market segmentation 🡪 ask algorithm to discover market segments automatically based on a database of customer data
* **Model Representation**
* Ex: Predicting housing prices w/ a data set of housing prices from Portland, Oregon w/ different sizes sold for a range of different prices.
* Someone is trying to sell a house of 1250 ft2 + predict how much they might be able to sell it for.
* 1 thing to do is fit a model, like a fit straight line
* 
* Based on that, maybe he can sell the house for around $220k = a supervised learning algorithm, b/c we're given the "right answer" for each examples.
* Told the actual size and price of each of house in the data used **regression** = predicting *a real-valued output* (price).
* The other most common type of supervised learning problem = **classification** = predict discrete-valued outputs (decide if tumors are malignant or benign = a zero-one valued discrete output)
* More formally, in supervised learning, we have a **training set**
* Ex: Training set of different housing prices 🡪 job = learn from data how to predict prices of houses
* **m** = sample size/observations, **x** = input variables/**features**, **y** = output/**target** variable to predict.
* **(x, y)** = a single training example/observation/row in a table
* **(x(i), y(i))** = ith training example. So this superscript i over here, this is not exponentiation right?
* Feed a training set to a ML algorithm whose job it to then output a function, which by convention is usually denoted **h** for hypothesis
* The job of the hypothesis function, h, is to take in the value of x + try to output the estimated value of y for the corresponding x = *h is a function that maps from x's to y's.*
* When designing a ML algorithm, the next thing to decide is how to *represent* this hypothesis h.
* 
* Means we are going to predict that y is a *linear function* of x to predict that y is some straight line function of x.
* Sometimes we'll want to fit more complicated, perhaps non-linear function, but since this linear case is a simple building block, we start w/ this 1st example of fitting linear functions
* This model = **linear regression** **w/ 1 variable** (x) = predicting all prices (y) as functions of 1 variable (x) = **univariate linear regression**.
* **The cost function** = lets us figure out how to fit the best possible straight line to data.
* In linear regression, we have a training set + a hypothesis used to make predictions via a linear function
* **Parameters** of a model = intercept (θ 0) and **regression coefficient** (slope or θ 1) .
* W/ different choices of the parameters θ 0 + θ 1, we get different hypothesis functions.
* In linear regression, we have a training set + we want to come up w/ values for the parameters so that the straight line we get corresponds to a straight line that somehow fits the data well
* The idea is to choose parameters so that h(x) (value predicted on input x) is *at least* *close* to the actual values y for the examples in the training set
* In our training set, we've given a number of x’s that decide a house size + we know the actual price it was sold for.
* We try to choose values for the parameters so that, at least in the training set, given x, we make reasonably accurate predictions for Y
* Linear regression as a **minimization problem 🡪** minimize the square difference between the output of the hypothesis, h(x), and the actual price of a house, y.
* Want to sum the square difference between (h(x(i)) and y(i)) over my training set and halve it + minimize that result

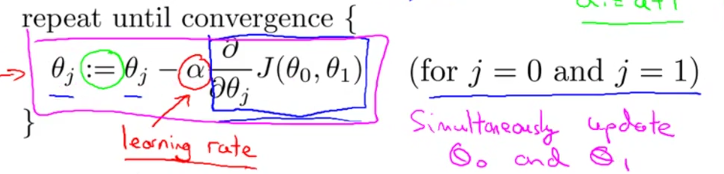
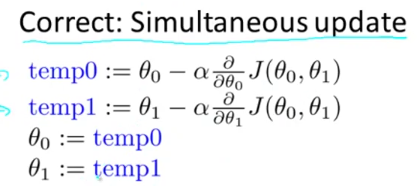
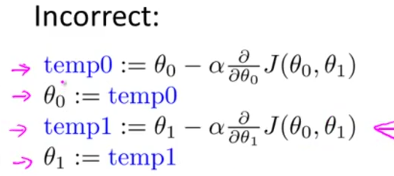
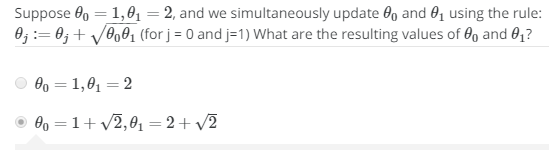
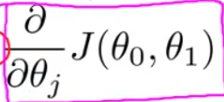
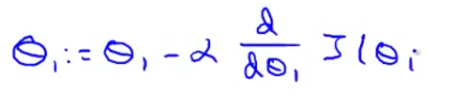
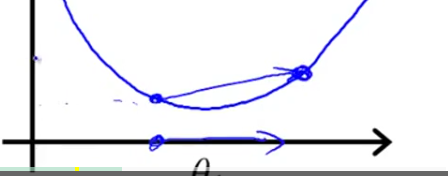


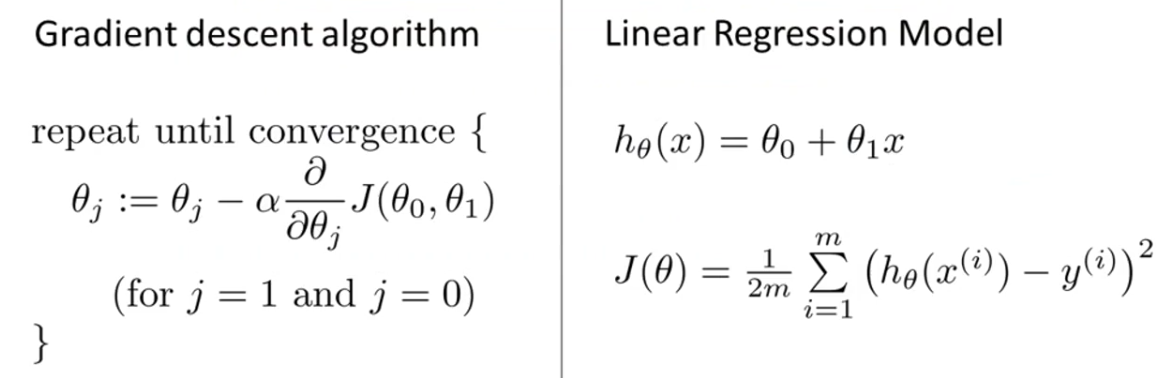
* So we want to find the values of θ 0 and θ 1 so that the average (1/2m) sum of square errors is minimized, giving the overall objective function for linear regression (below in pink)
* 
* **A cost function (square error)**
* 
* 
* The squared error cost function is a reasonable choice and works well for most regression problems.
* There are other cost functions that work pretty well, but the square cost function is probably the most commonly used one for regression problems.
* To summarize:
* **Hypothesis = h(x) = θ0 + θ1\*x**
* **Parameters = θ0 and θ1**
* **Cost Function = J(θ0,θ1) = 1/2m\*sum(from i to m) of (h(x(i)) – y(i))^2**
* **Goal = minimize (θ0,θ1) J(θ0,θ1)**
* It turns out the 2 key functions we want to understand are the hypothesis function + a cost function.
* The hypothesis, for a fixed value of θ1, is a function of X (size of a house).
* In contrast, the cost function, J, is a function of the parameter, θ1, which controls the *slope* of the straight line.

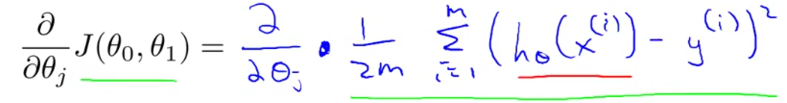
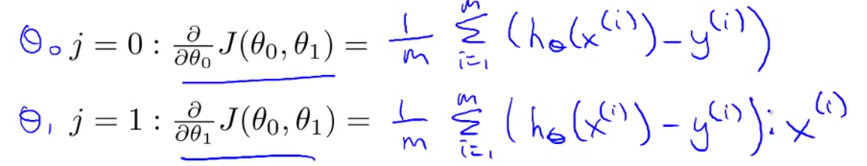
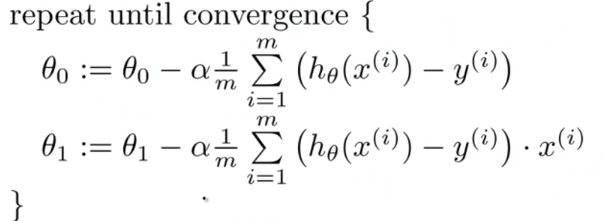
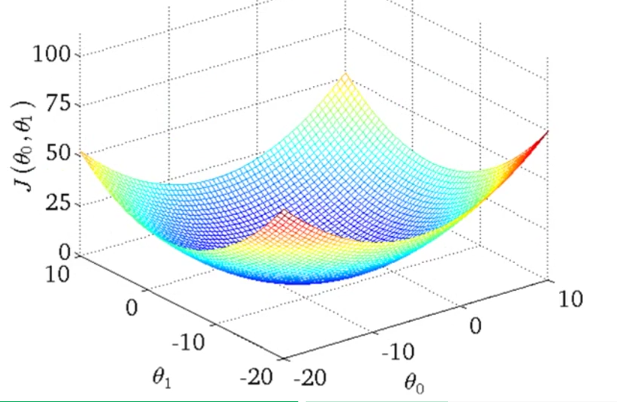


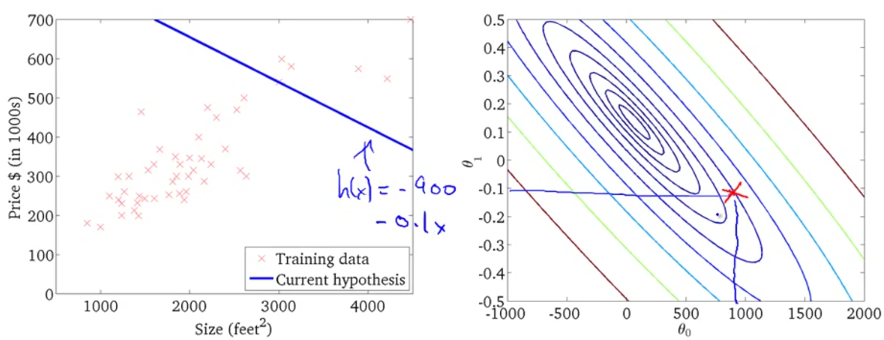
* W/ h(x), we have 3 points at, and when θ1 = 1, then my hypothesis look like the above
* Now to figure out what is J(θ1) when θ1 = 1 by computing the cost function for 1.
* 
* So it turns out to be the sum of (1-1)^2 + (2-2)^2 + (3-3)^2 = 0 b/c if θ1 = 1, h(x(i)) = y(i) exactly
* So, we know now that J(1) = 0
* 
* Θ-1 can take on a range of different values 🡪 negative values, zero, positive values.
* So if θ-1 = 0.5 🡪 slope = 0.5
* 
* J(0.5) = 1/2m \* (0.5 – 1)^2 + (1 – 2)^2 + (1.5 – 3)^2 = (1/(2\*3))\*3.5) = 3.5/6 = 0.58
* J(0) = 1/2m \* (0 – 1)^2 + (0 – 2)^2 + (0 – 3)^2 = (1/(2\*3))\*14) = 14/6 = 2.3
* 
* It turns out you can have negative values of θ1 as well, then h(x) = a hypothesis w/ a negative slope
* You can actually keep on computing these errors, and -0.5 turns out to have really high error, ~5.25.
* By computing that range of values, you can actually slowly create plot out J(θ1)
* 
* To recap, each value of θ1 corresponds to a different h(x)/a different straight line fit
* For each value of θ1, we wound up w/ a different value of J(θ1) + used it to trace out its plot
* Remember, the optimization objective for our ML algorithm is to choose the value of θ1 that minimizes J(θ1)
* Looking at this curve, the value that minimizes J(θ1) is 1, which is indeed the best possible straight line fit through our data, just for this particular training set
* Again, we want to understand the hypothesis H and the cost function J.
* Set θ0 = 50 and θ1 = 0.06 to end up w/ h(x) = 50 + 0.06x
* Given these value of the parameters, we want to plot the corresponding cost, but now w/ 2 parameters the plot gets a little more complicated
* It still has a similar bow shape, and, in fact, depending on the training set, you might get a cost function that looks like a 3-D surface plot:
* As you vary the 2 parameters, you get different values of the cost function J (θ0,θ1) and the height of the surface above a particular point of θ0, θ1 (the x-axis value), and the height of the surface of the points indicates the value of J of θ0, θ1.
* 
* **Contour Figures/Plots** (on right) w/ axis = θ0 and θ1
* 
* Each of ellipse shows a set of points that takes on the same value for J(θ0, θ1).
* So concretely take 3 points on the magenta ellipse + they all have the same value for J (θ0, θ1).
* 
* A contour figures is a more convenient way to visualize J.
* E: θ0 = 800, θ1 = -0.15, and this point on the right corresponds to that hypothesis on the left (intersects y-axis at ~800 w/ a slope of ~-0.
* 
* This line is really not such a good fit to the data, + you find that it's cost (red x point on right) is a value that's pretty far from the minimum (middle of smallest ellipse in the middle)
* So this is a pretty high cost b/c its not a good fit to the data.
* 
* This is a different hypothesis that's still not a great fit for the data but may be slightly better w/ θ0 = 360 + θ1 = zero 🡪 h(x) = 360 + 0\*x
* This hypothesis again has some cost plotted as the height of J at that point.
* 
* This is actually not quite at the minimum, but is pretty close, so this is not such a bad fit to the data
* The sum of squared errors = sum of squared distances 🡪 so of all of these errors, this h(x) error is pretty close to the minimum
* What we really want is an efficient algorithm/piece of software for automatically finding the values of θ0 + θ1 that minimizes the cost function
* **Gradient Descent** = algorithm for minimizing the cost function J.
* is a more general algorithm, and is used not only in linear regression, but all over in ML. And
* Assume we have some function J(θ 0, θ 1) + we want an algorithm for minimizing it
* Just as an aside it turns out that gradient descent actually applies to more general functions.
* i.e. minimize over θ0…..θ(n) the value of J(θ0…..θ(n))
* 
* The idea for gradient descent is to start w/ some initial guesses for θ 0 + θ 1 (common choice 🡪θ 0 = 0, θ 1 = 0) and then keep changing them a little bit to try to reduce J(θ 0, θ 1), until hopefully, we wind up at a minimum, or a local minimum.
* 
* Θ 0 + θ 1 = horizontal axes, J = vertical axis 🡪 height of the surface shows J + we want to minimize it.
* Start off w/ θ 0, θ 1 at some point on the surface
* Imagine that this figure shows two hills + you are physically standing at the point on the hill
* In gradient descent, we're going to spin 360 degrees + look around + ask “if I were to take a little baby step in some direction, + I want to go downhill as quickly as possible, what direction do I take?”
* To go down, want to physically walk down this hill as rapidly as possible.
* Then you take that direction + you're at a new point on your hill + again, look all around + see what direction to step in order to take a little baby step downhill + take another step in that direction + then keep going until you converge to a **local minimum**.
* 
* This 1st time we ran gradient descent we started at a point + now imagine we initialized gradient descent just a couple steps to the right instead, + then repeat the process from that point, gradient descent would've taken you to a 2nd local minimum, more to the right.
* 
* This is a interesting property of gradient descent
* This is the definition of the gradient descent algorithm.

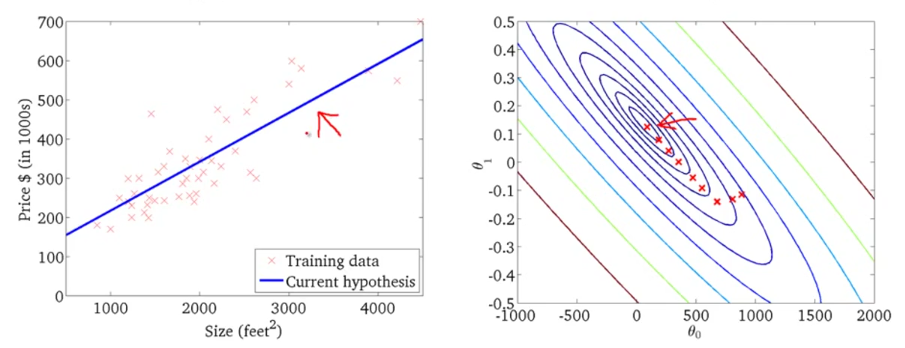
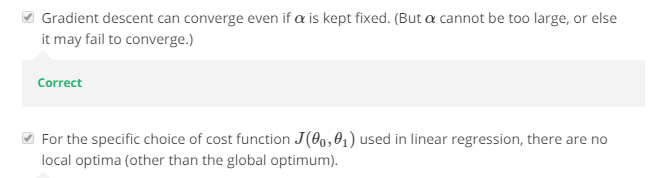


* Repeatedly do this until **convergence** 🡪 update θ j by taking θ j + subtracting from it alpha\*a term
* **:=** 🡪 denotes assignment
* a := b 🡪 in a CPU, this means take the value in b + use it to overwrite whatever value is a.
* “set a to be equal to the value of b”
* Whereas in contrast, if I use the equal sign + write a = b, this is a **truth assertion.**
* **Alpha** = a number called the **learning rate**.
* controls how big a step we take downhill w/ gradient descent.
* If alpha is very large = very aggressive gradient descent procedure = take huge steps downhill
* If alpha is very small = taking baby steps downhill.
* Then we have a **derivative term**
* 
* There's 1 more subtlety about gradient descent 🡪 in gradient descent we're going to update θ 0 + θ 1, which takes place “for j = 0 and j = 1”
* So you're update θ 0 + θ 1 + for this update equation, you want to *simultaneously update* θ 0 + θ 1.
* Meaning in this equation, we're going update θ 0 := …. + update θ 1 := …..
* The way to implement this is to compute the right-hand for θ 0 + θ 1 + then simultaneously update θ 0 and θ 1
* We set *temp0* + *temp1* to compute the right-hand sides + store them into variables temp0 + temp1 + then update θ 0 + θ 1 simultaneously = the correct implementation.
* 
* 
* In the second attempt, by the time updated θ 0 (2nd step), then you would be using the new value of θ 0 in J of the 3rd step, which gives a different value of temp1
* It turns out the way gradient descent is usually naturally implemented is w/ simultaneous updates.
* When people talk about gradient descent, they always mean simultaneous update
* If you implement the non-simultaneous update, it will probably work anyway, but this algorithm wasn't right + is some other algorithm w/ different properties that can, for various reasons, behave in slightly stranger ways
* 
* **Alpha/Learning Rate** controls how big a step we take when updating the parameter theory(j).
* The **derivative term**
* 
* Say we have a cost function, j of just 1 parameter, θ1, where θ1 = a real number + let's try to understand what gradient decent would do on this function.
* 
* So we initialize gradient descent w/ θ1 to the right, and gradient descent will update Θ1 as θ1 – (alpha\*derivative of J(θ1) w/ respect to θ1)
* 
* We compute the derivative/take the tangent to the current J(θ1) + look at the slope of this line
* Slope = positive = positive derivative 🡺 so my update ( **:=** )to θ1 is going to be θ1 – alpha\* some positive number (*Alpha is always a positive number*)
* So we end up w/ θ one – a positive = move new θ value to the left = decrease θ1,
* We want to head in this direction 🡪 towards minimum J(θ1)
* If the initial θ1 was to the left, we’d have a negative tangent slope = negative derivative = θ1 – negative # = increase θ1 = update θ1 to the right towards the minimum J(θ1)
* If alpha is too small we multiply updated θ by some small number = taking a small step = will need a lot of steps to get to the minimum = gradient descent is
* If alpha is too large= gradient descent can *overshoot* the minimum and may even *fail* to converge or even actually diverge
* 
* Can end up w/ a worse value of θ
* Can continue to overshoot actually get further + further away from the minimum.
* So if alpha is to large, it can fail to converge or even diverge.
* What if your parameter θ 1 is *already at a local minimum*?
* Initialize 1 at a local minimum/optimum 🡪 derivative = 0 = tangent slope = 0 🡪 derivative term = 0 🡪 gradient descent update = θ1 – 0
* If you're already at the local optimum, it leaves θ 1 = gradient descent does absolutely nothing
* This also explains why gradient descent can converge to the local minimum even w/ a fixed learning rate/alpha.
* Imagine cost function J of θ + start gradient descent w/ a steep derivative 🡪 next step = derivative = less steep b/c as I approach the minimum, my derivative gets closer + closer to 0
* Now naturally take a smaller step + now even closer to global minimum = derivative = even smaller
* As gradient descent runs, you will automatically take smaller and smaller steps until eventually you're taking very small steps + finally converge to the to the local minimum.
* So as we approach local minimum, the *derivative term* will automatically *get smaller*, so gradient descent will automatically take smaller steps = no need to decrease alpha.
* The gradient descent algorithm can be used to try to minimize any cost function J
* Put together gradient descent w/ a cost function = gives an algorithm for linear regression



* Want to *apply gradient descent to minimize squared error/cost function*.
* In order to apply gradient descent, the key term needed = the *derivative term* over here 🡪plug in the definition of J
* 
* Then plug in the definition for hypothesis(x(i))
* 
* Turns out we need to figure out the partial derivative for 2 cases 🡪 for J = 0 and J = 1.
* 
* Computing these partial derivative terms requires some multivariate calculus
* So armed w/ these definitions, we worked the derivatives = the slope of the cost function J + can now plug them back in to our gradient descent algorithm.
* **Gradient Descent For Linear Regression**
* 
* W/ alpha times the partial derivatives w/ respect to θ0 and respect to θ1 worked out before
* REMINDER: you should be implementing this so the update θ 0 + θ 1 are simultaneous
* 1 of the issues we saw with gradient descent is it can be susceptible to local optima = depending on where you initialize it, you can end up at different local optima.
* It turns out the cost function for linear regression is *always going to be a bow shaped function =* a **convex function**.
* 
* This function only has 1 global optimum and no local optima
* Gradient descent on this type of cost function (which you get whenever using linear regression) will always converge to the global optimum (b/c there are no other local optimum)
* Now, we usually initialize parameters at (0,0), but for the demonstration we’ll do θ0 = 900 + θ1 = ~-0.1 🡪 corresponds to h(x)= 900-0.1x



* Now, if we take 1 step in gradient descent, we move down and left + the h(x) lines changes a bit
* As I take further steps of gradient descent, we go down in cost (move parameter data point close and closer to middle/minimum) and h(x) line better and better fits to the data
* Eventually we wind up at the global minimum = gets a good fit to the data set of housing prices
* 
* Can now use it to predict the price to sell a 1250 square ft. house = ~$250k
* The algorithm just used is sometimes called **batch gradient descent** (ML people were not always great at giving names to algorithms)
* *Batch Gradient Descent* refers to the fact that in every step of gradient descent, we're looking at ALL training examples 🡪 when computing the derivatives, we're computing the sums over our “m” training examples 🡪 refers to the fact that we're looking at the *entire batch* of training examples
* There are sometimes other versions of gradient descent that are not batch versions + do not look at the entire training set, but instead look at small subsets of the training set at a time.
* If you've seen advanced linear algebra before, you might know that there exists a solution for numerically solving for the minimum of the cost function J w/out needing to use an iterative algorithm like gradient descent = the **normal equations method**.
* But gradient descent will scale better to larger data sets than that normal equation method, + we can use it in lots of different contexts + in lots of different ML problems.
* 

1. A computer program is said to learn from experience **E** with respect to some task **T** and some performance measure **P** if its performance on **T**, as measured by **P**, improves with experience **E**. Suppose we feed a learning algorithm a lot of historical weather data, and have it learn to predict weather. What would be a reasonable choice for P?

* **The probability of it correctly predicting a future date's weather.**

1. The amount of rain that falls in a day is usually measured in either millimeters (mm) or inches. Suppose you use a learning algorithm to predict how much rain will fall tomorrow. Would you treat this as a classification or a regression problem?

* **Regression**

1. Suppose you are working on stock market prediction, and you would like to predict the price of a particular stock tomorrow (measured in dollars). You want to use a learning algorithm for this. Would you treat this as a classification or a regression problem?

* **Regression**

1. Some of the problems below are best addressed using a supervised learning algorithm (*we teach CPU*), and the others with an unsupervised learning algorithm (*CPU teaches itself)*. In each case, assume some appropriate dataset is available for your algorithm to learn from.

* Given data on crop yields over the last 50 years, learn to predict next year's yields 🡪 **Supervised**
* Given data on how 1000 patients respond to an experimental drug (effectiveness of treatment, side effects, etc.), discover whether there are different categories or "types" of patients in terms of how they respond to the drug, and if so what these categories are. 🡪 **Unsupervised**
* Examine a web page, and classify whether the content on the page should be considered "child friendly" (e.g., non-pornographic, etc.) or "adult." 🡪 **Supervised**
* Given a large dataset of medical records from patients suffering from heart disease, try to learn whether there might be different clusters of such patients for which we might tailor separate treatments. 🡪 **Unsupervised**

1. Which of these is a reasonable definition of machine learning?

* **Machine learning is the field of study that gives computers the ability to learn without being explicitly programmed.**

1. An email program watches which emails we mark as spam or not spam, and based on this activity learns how to better filter spam. What is the task, T, here?

* **Classify emails as spam or not spam**
* **E = watching you classify spam and not spam**
* **P = probability the program will correctly ID /not spam, OR the # or fraction/% of emails correctly classified as spam/not spam**

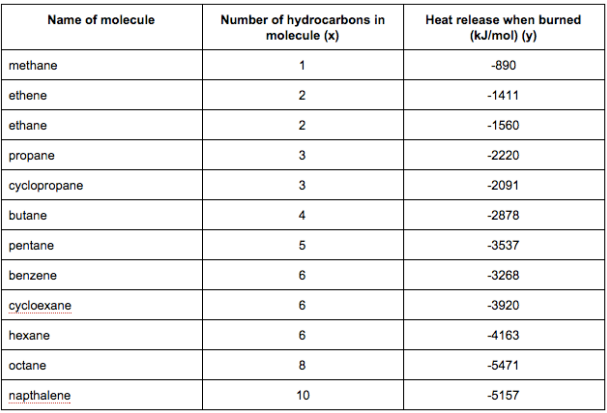
1. Consider the problem of predicting how well a student does in her 2nd year of college, given how well she did in her 1st year. Specifically, let x = # of "A" grades (including A-. A, A+) a student receives in their 1st year of college. We would like to predict the value of y = the # of "A" grades they get in their 2nd year. Here each row is 1 training example. Recall that in linear regression, our hypothesis is hθ(x)=θ0+θ1x + we use m to denote the # of training examples.



For the training set given above, what is the value of m?

* **4**

1. Many substances that can burn (such as gasoline + alcohol) have a chemical structure based on carbon atoms + for this reason they’re called hydrocarbons. A chemist wants to understand how the # of carbon atoms in a molecule affects how much energy is released when that molecule combusts (is burned). The chemist obtains the dataset below. In the column on the right, “kJ/mol” is the unit measuring the amount of energy released.



You would like to use linear regression (hθ(x)=θ0+θ1x) to estimate the amount of energy released (y) as a function of the # of carbon atoms (x). Which of the following do you think will be the values you obtain for θ0 and θ1? You should be able to select the right answer without actually implementing linear regression.

* **θ0=−569.6,θ1=−530.9**

1. Suppose we set θ0=−1,θ1=0.5. What is hθ(4)?

* **-1 + (0.5\*5) = 1**

1. Let f be some function so that f(θ0,θ1) outputs a #. For this problem, f is some arbitrary/unknown smooth function (not necessarily the cost function of linear regression, so f may have local optima). Suppose we use gradient descent to try to minimize f(θ0,θ1) as a function of θ0 and θ1. Which of the following statements are true?

* **If θ0 and θ1 are initialized at a local minimum, then one iteration will not change their values.**
* **If the learning rate is too small, then gradient descent may take a very long time to converge.**
* **If θ0 and θ1 are initialized at the global minimum, then one iteration will not change their values.**
* **If the first few iterations of gradient descent cause f(θ0,θ1) to increase rather than decrease, then the most likely cause is that we have set the learning rate α to too large a value.**

1. Suppose that for some linear regression problem (say, predicting housing prices as in the lecture), we have some training set, and for our training set we managed to find some θ0, θ1 such that J(θ0,θ1)=0. Which of the statements below must then be true?

* **Our training set can be fit perfectly by a straight line**
* **We can perfectly predict the value of y even for new examples that we have not yet seen. (e.g., we can perfectly predict prices of even new houses that we have not yet seen.)**

1. Consider the following training set of m=4 training examples:

x y

1 0.5

1. 1

4 2

0 0

Consider the linear regression model hθ(x)=θ0+θ1x. What are the values of θ0 and θ1 that you would expect to obtain upon running gradient descent on this model? (Linear regression will be able to fit this data perfectly.)

* **θ0=0,θ1=0.5**

1. Suppose we set θ0=0,θ1=1.5 in the linear regression hypothesis from Q1. What is hθ(2)?

* **3**