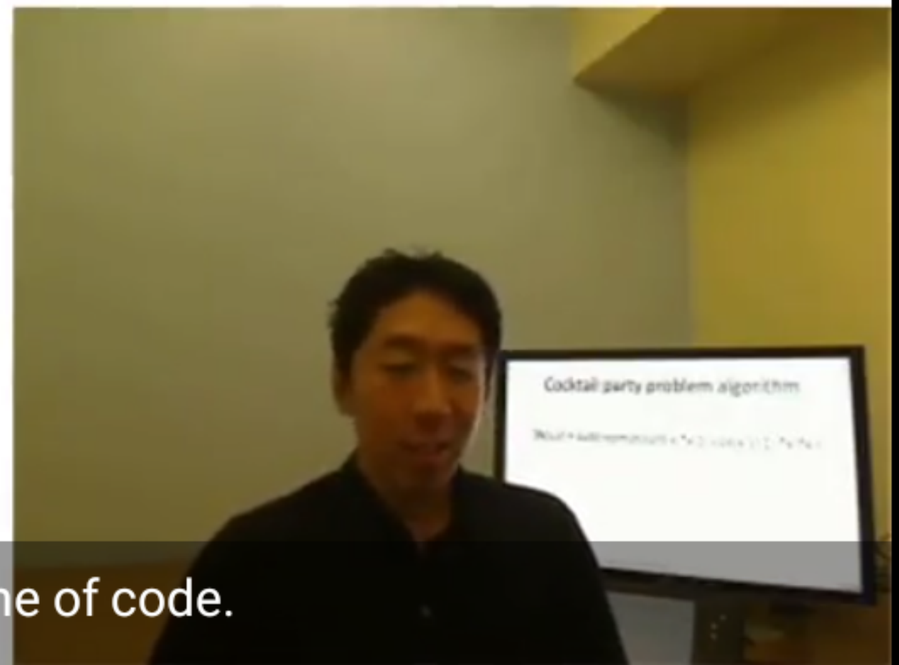
Cocktail party problem algorithm

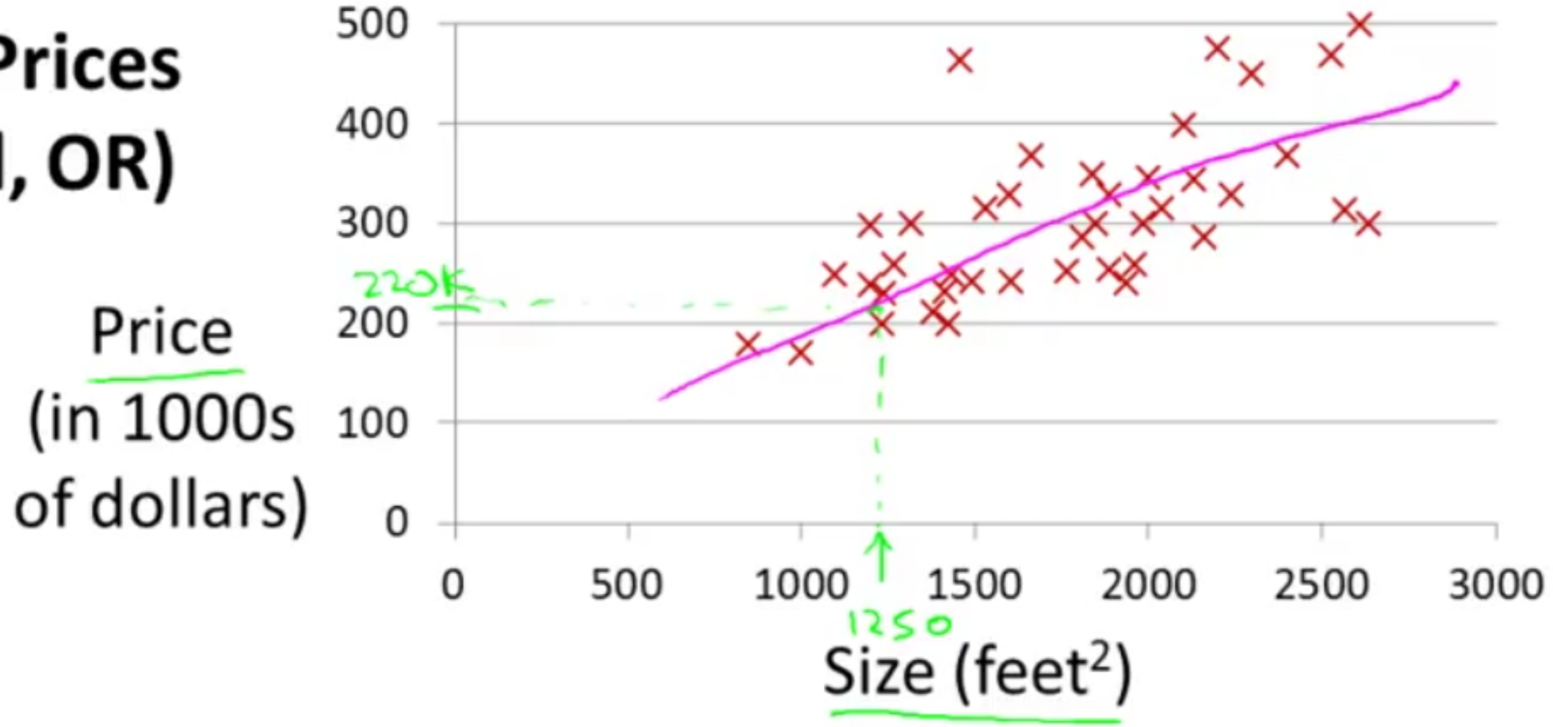
```
[W,s,v] = svd((repmat(sum(x.*x,1),size(x,1),1).*x)*x');
```



[Source: Sam Roweis, Yair Weiss & Eero Simoncelli]

time to come up with this line of code.

Housing Prices (Portland, OR)



Supervised Learning

Given the "right answer" for each example in the data.

Regression Problem

Predict real-valued output

the term regression refers to the fact
that we are predicting a real-valued output

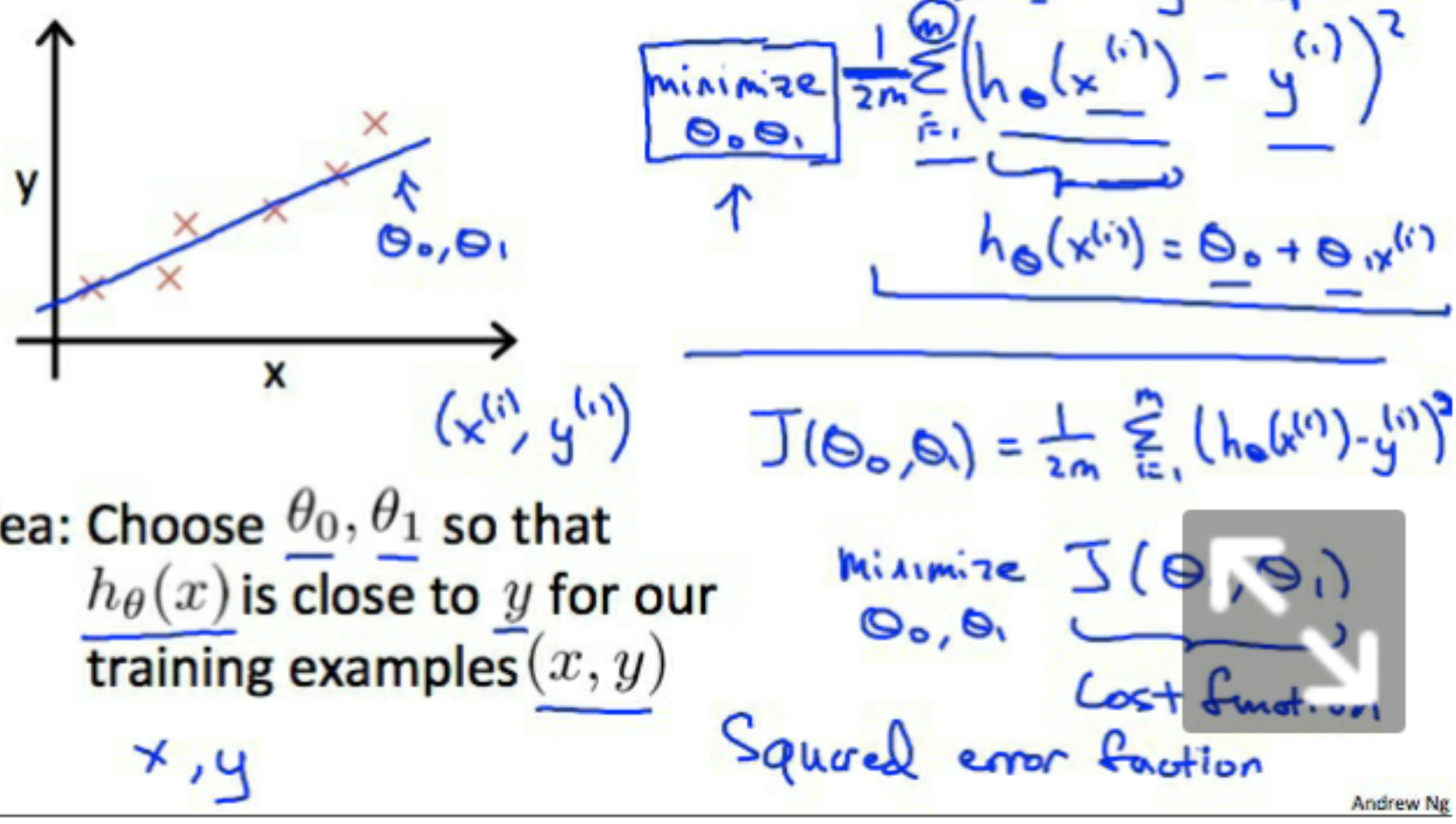
Cost Function

We can measure the accuracy of our hypothesis function by using a **cost function**. This takes an average difference (actually a fancier version of an average) of all the results of the hypothesis with inputs from x's and the actual output y's.

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

To break it apart, it is $\frac{1}{2}\ \bar{x}$ where $\bar{x}$ is the mean of the squares of $h_\theta(x_i) - y_i$, or the difference between the predicted value and the actual value.

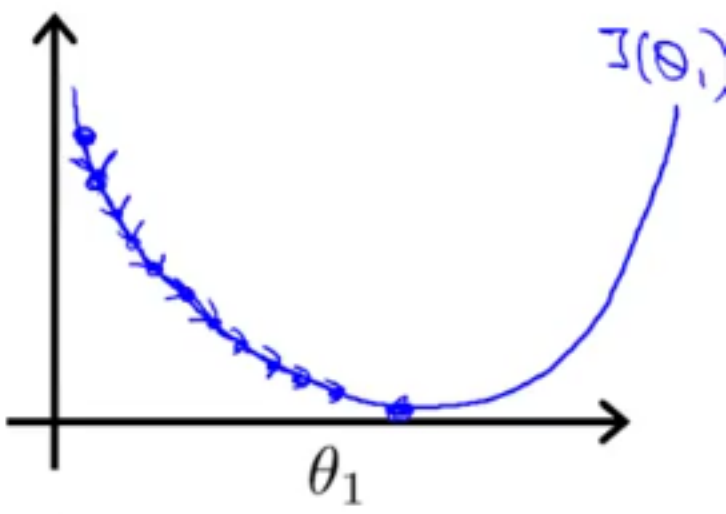
This function is otherwise called the "Squared error function", or "Mean squared error". The mean is halved $\left(\frac{1}{2}\right)$ as a convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the $\frac{1}{2}$ term. The following image summarizes what the cost function does:



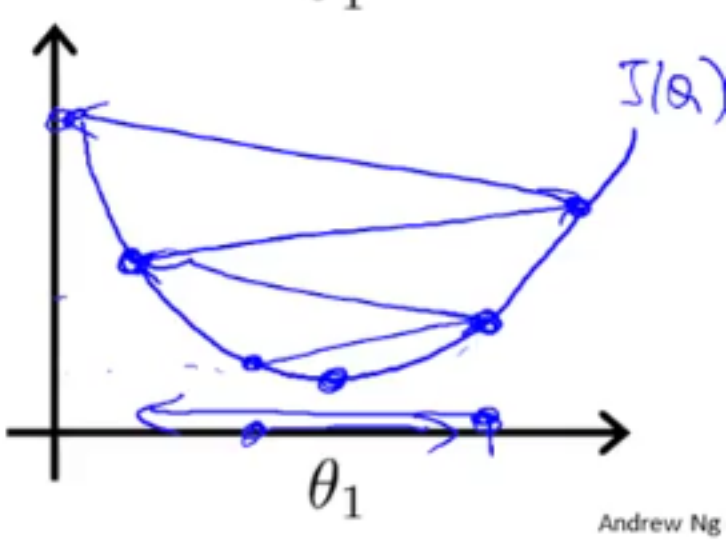
Gradient Descent Intuition

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

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.



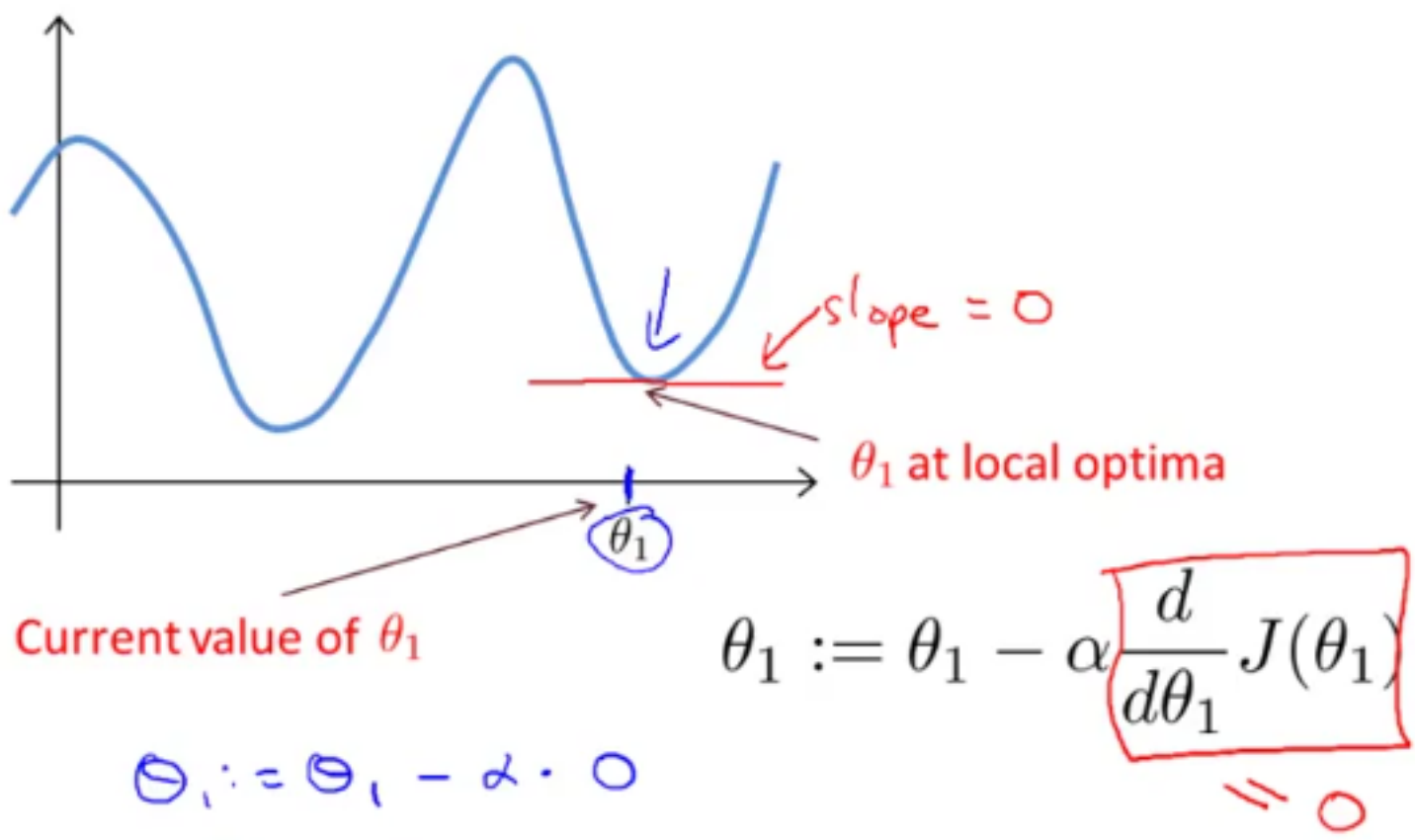
Andrew Ng

Now, I have another question for you. So this a tricky one and when I was first learning this stuff it actually took me a long time to figure this out. What if your parameter theta 1 is already at a local minimum, what do you think one step of gradient descent will do?

So let's suppose you initialize theta 1 at a local minimum. So, suppose this is your initial value of theta 1 over here and is already at a local optimum or the local minimum. It turns out the local optimum, your derivative will be equal to zero.

So for that slope, that tangent point, so the slope of this line will be equal to zero and thus this derivative term

Gradient Descent Intuition



So for that slope, that tangent point, so the slope of this line will be equal to zero and thus this derivative term is equal to zero. And so your gradient descent update, you have theta one cuz I updated this theta one minus alpha times zero.

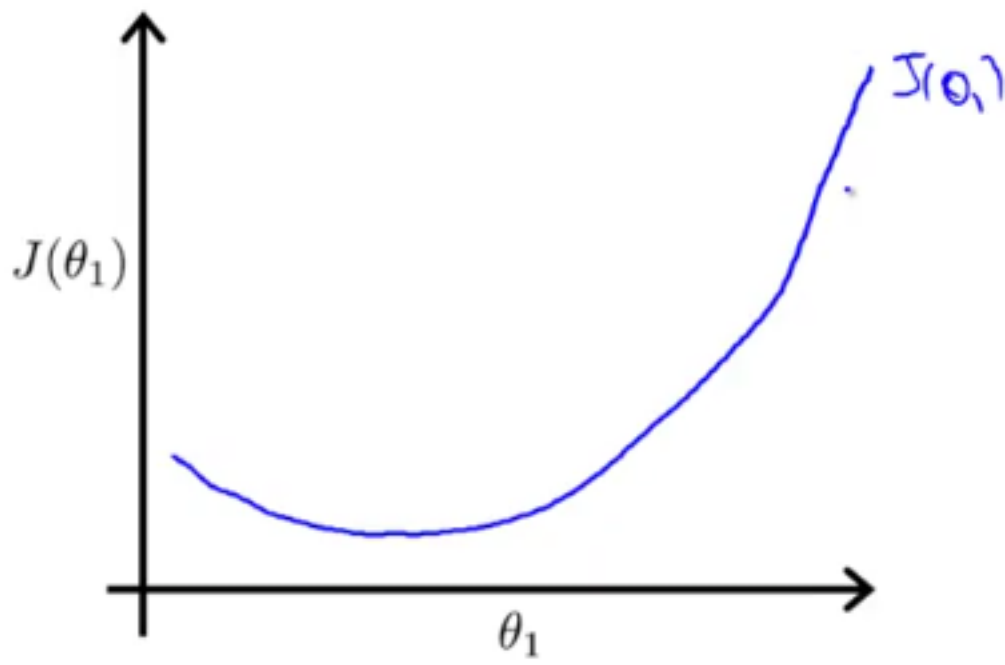
And so what this means is that if you're already at the local optimum it leaves theta 1 unchanged cause its updates as theta 1 equals theta 1.

So if your parameters are already at a local minimum one step with gradient descent does absolutely nothing it doesn't your parameter which is what you want because it keeps your

Gradient Descent Intuition

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

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



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

Andrew Ng

So here's a cost function J of theta that maybe I want to minimize and let's say I initialize my algorithm, my gradient descent algorithm, out there at that magenta point. If I take one step in gradient descent, maybe it will take me to that point, because my derivative's pretty steep out there.

Right? Now, I'm at this green point, and if I take another step in gradient descent, you notice that my derivative, meaning the slope, is less steep at the green point than compared to at the magenta point out there.

Because as I approach the minimum, my derivative gets closer and closer

"Batch" Gradient Descent

"Batch": Each step of gradient descent uses all the training examples.

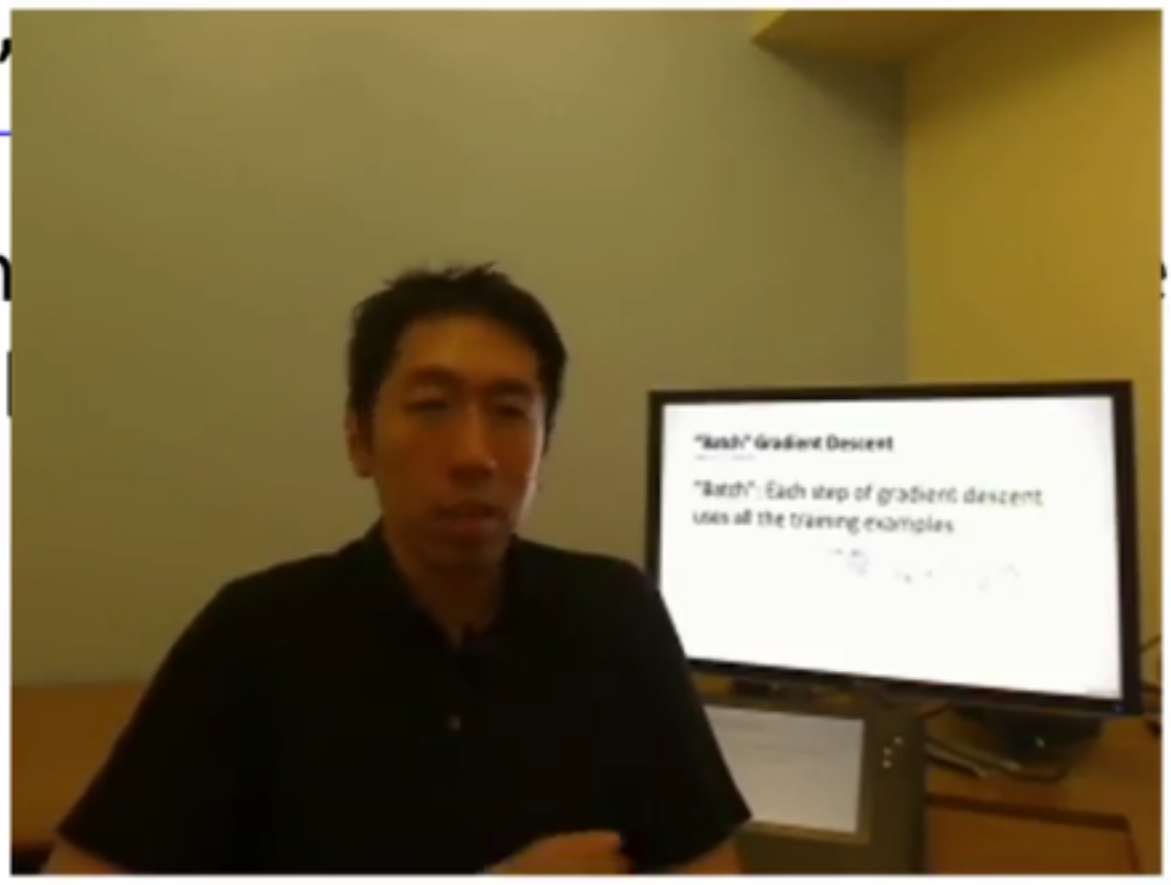
Finally just to give this another name it turns out that the algorithm that we just went over is sometimes called batch gradient descent. And it turns out in machine learning I don't know I feel like us machine learning people were not always great at giving names to algorithms.

But the term batch gradient descent refers to the fact that in every step of gradient descent, we're looking at all of the training examples. So in gradient descent, when computing the derivatives, we're computing the sums [INAUDIBLE].

So ever step of gradient descent we end up computing something like

“Batch”

“Batch nt
uses a



that method as well that just solves for the minimum of the cost function j without needing these multiple steps of gradient descent. That other method is called the normal equations method.

But in case you've heard of that method it turns out that gradient descent will scale better to larger data sets than that normal equation method.

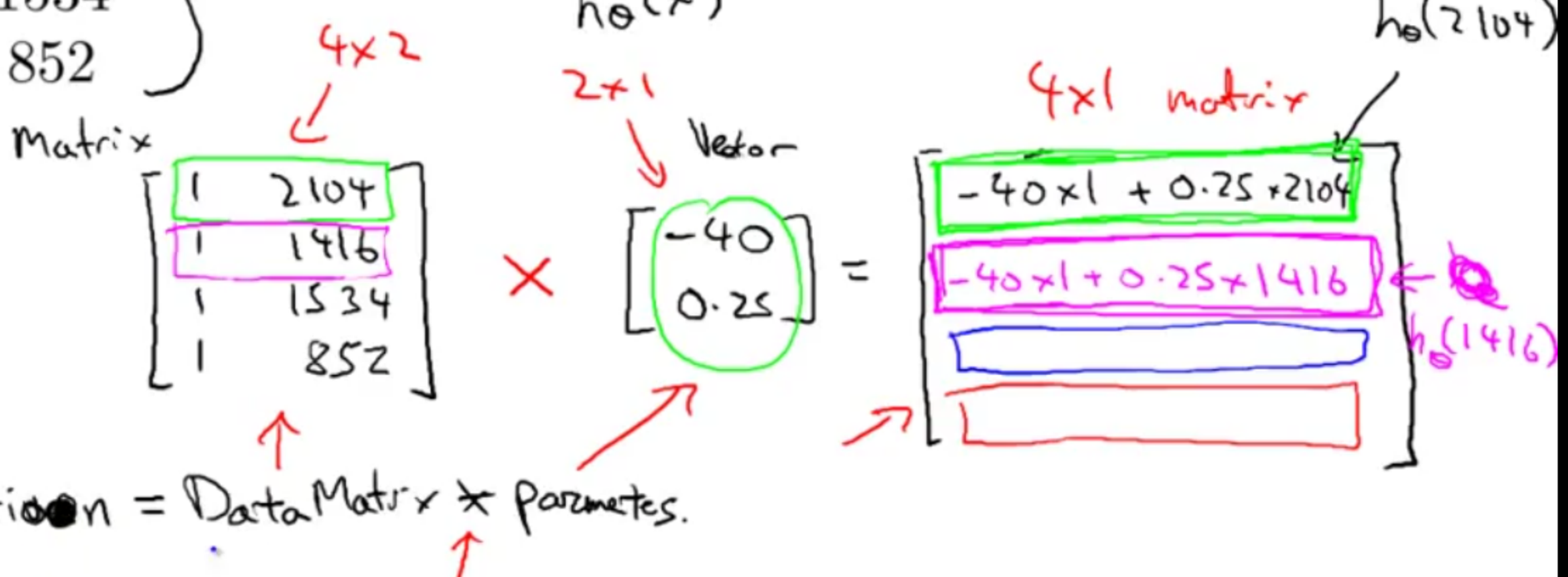
And now that we know about gradient descent we'll be able to use it in lots of different contexts and we'll use it in lots of different machine learning problems as well. So congrats on learning about your first machine

House sizes:

→ 2104
→ 1416
→ 1534
→ 852

$$h_\theta(x) = -40 + 0.25x$$

$h_\theta(x)$



prediction = DataMatrix * parameters.

just implement this one

Andrew Ng

House sizes:

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

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

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

$\times$

Matrix

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

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

can do that very efficiently using
a matrix-matrix multiplication.

Andrew Ng

1 = "identity."

$3 \cdot (3^{-1}) = 1$ ($3^{-1} = \frac{1}{3}$)

$12 \times (12^{-1}) = 1$ ($12^{-1} = \frac{1}{12}$)

$0 \cdot (0^{-1})$ undefined

Not all numbers have an inverse.

Matrix inverse:

If A is an m x m matrix (square matrix, #rows = #columns), and if it has an inverse, A^{-1}

$$A = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$A(A^{-1}) = A^{-1}A = I.$$

E.g.

$$\begin{bmatrix} 3 & 4 \\ 2 & 16 \end{bmatrix} \begin{bmatrix} 0.4 & -0.1 \\ -0.05 & 0.075 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I_{2\times 2}$$

A (2×2) $\quad A^{-1} \quad A^{-1}A$

Matrices that don't have an inverse are "singular" or "degenerate"

Gradient Descent

Previously (n=1):

Repeat {

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

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

(simultaneously update θ_0, θ_1)

}

New algorithm $(n \geq 1)$:

Repeat {

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

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

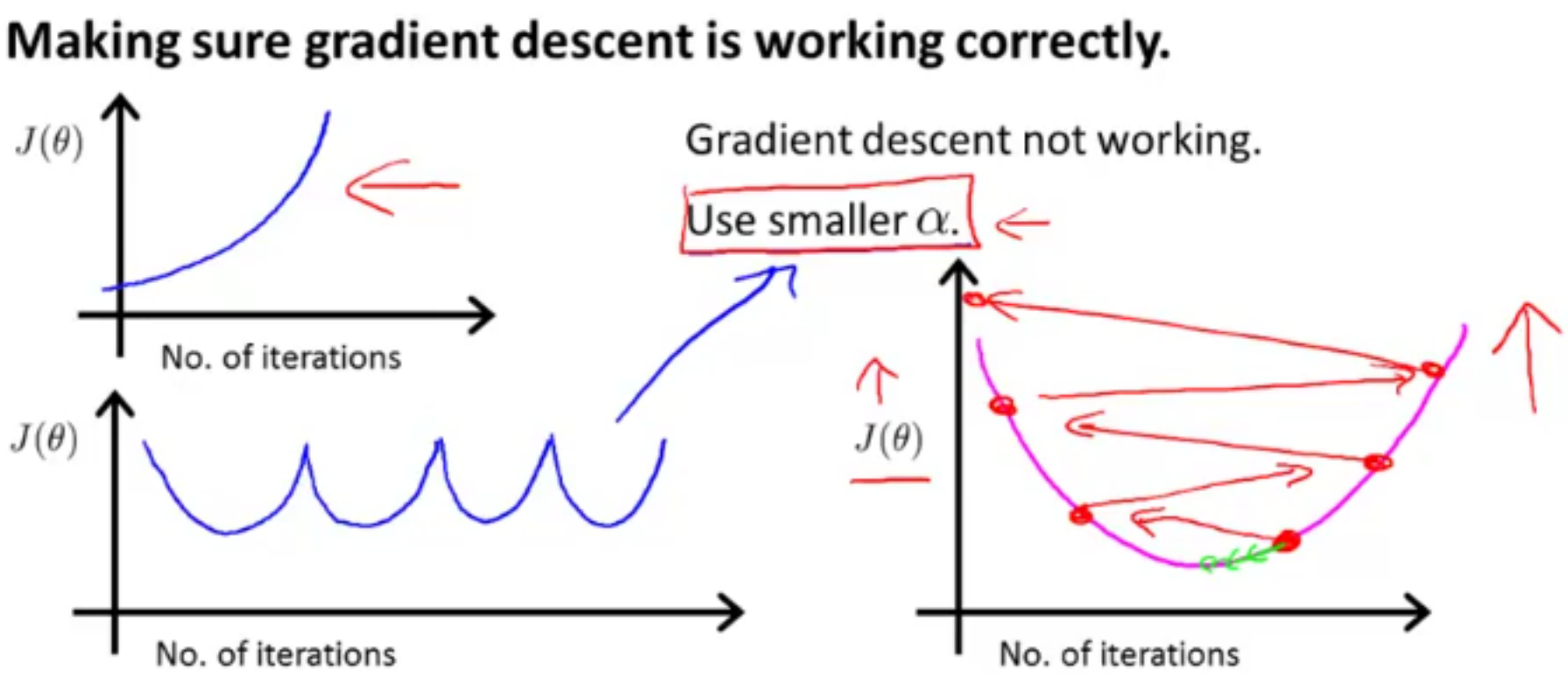
}

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

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

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

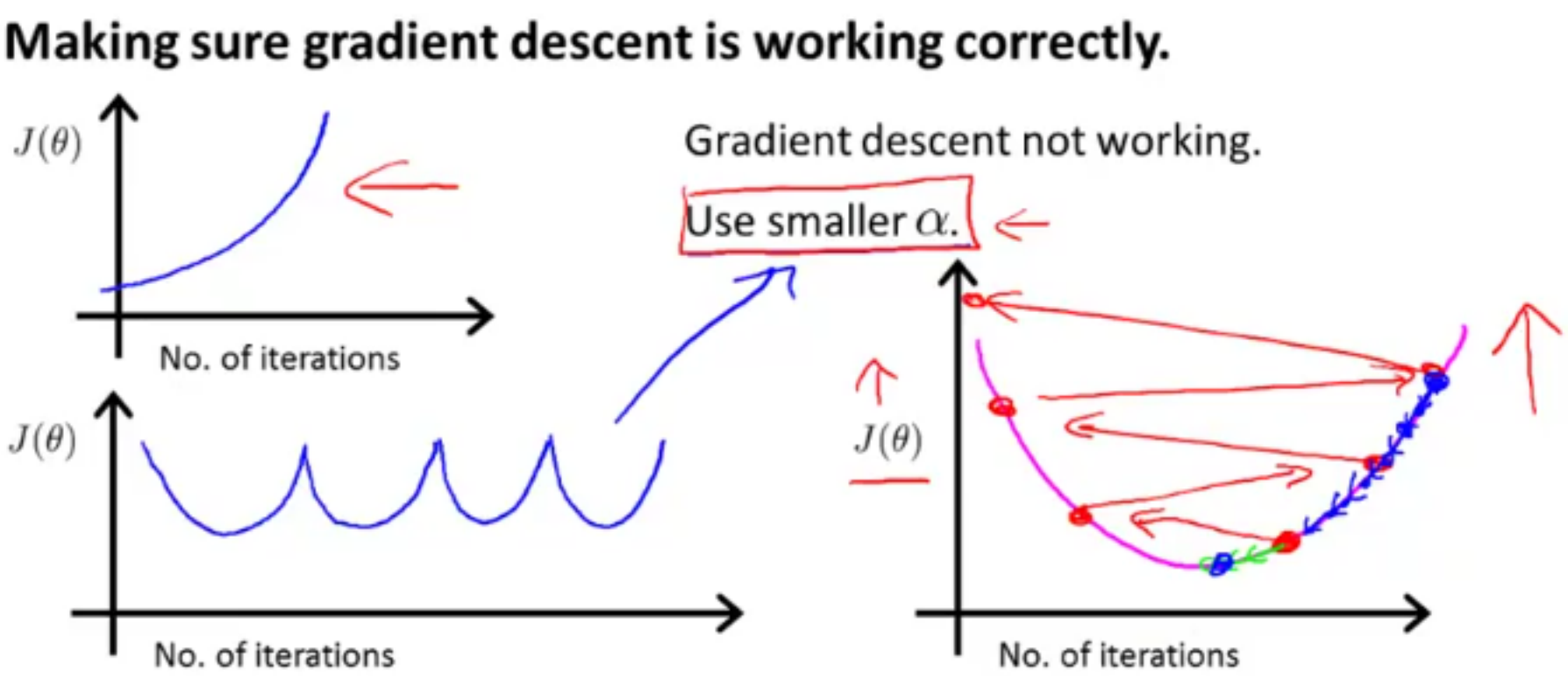
$\ldots$



Similarly sometimes you may also see J(theta) do something like this, it may go down for a while then go up then go down for a while then go up go down for a while go up and so on. And a fix for something like this is also to use a smaller value of alpha.

I'm not going to prove it here, but under other assumptions about the cost function J, that does hold true for linear regression, mathematicians have shown that if your learning rate alpha is small enough, then J(theta) should decrease on every iteration.

So if this doesn't happen probably means the alpha's too big, you should



And if alpha were too small, you might end up starting out here, say, and end up taking just minuscule baby steps. And just taking a lot of iterations before you finally get to the minimum, and so if alpha is too small, gradient descent can make very slow progress and be slow to converge.

To summarize, if the learning rate is too small, you can have a slow convergence problem, and if the learning rate is too large, J(theta) may not decrease on every iteration and it may not even converge. In some cases if the learning rate is too large,

Summary:

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

To choose α, try

$$\ldots, 0.001, \quad , 0.01, \quad , 0.1, \quad , 1, \ldots$$

Andrew Ng

To summarize, if the learning rate is too small, you can have a slow convergence problem, and if the learning rate is too large, J(theta) may not decrease on every iteration and it may not even converge. In some cases if the learning rate is too large, slow convergence is also possible.

But the more common problem you see is just that J(theta) may not de-crease on every iteration. And in order to debug all of these things, often plotting that J(theta) as a function of the number of iterations can help you figure out what's going on.

Concretely, what I actually do when I run gradient descent is I would try a

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

$$\ldots, 0.001, \quad , 0.01, \quad , 0.1, \quad , 1, \ldots$$

Andrew Ng

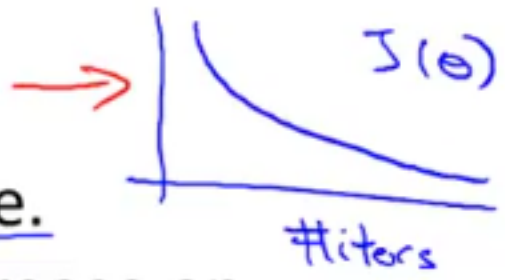
But the more common problem you see is just that J(theta) may not decrease on every iteration. And in order to debug all of these things, often plotting that J(theta) as a function of the number of iterations can help you figure out what's going on.

Concretely, what I actually do when I run gradient descent is I would try a range of values. So just try running gradient descent with a range of values for alpha, like 0.001 and 0.01. So these are factor of ten differences.

And for these different values of alpha are just plot J(theta) as a function of number of iterations, and then pick

Summary:

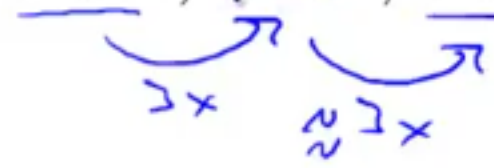
- If α is too small: slow convergence.
- If α is too large: $J(\theta)$ may not decrease on every iteration; may not converge.



(Slow converge also possible)

To choose α, try

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

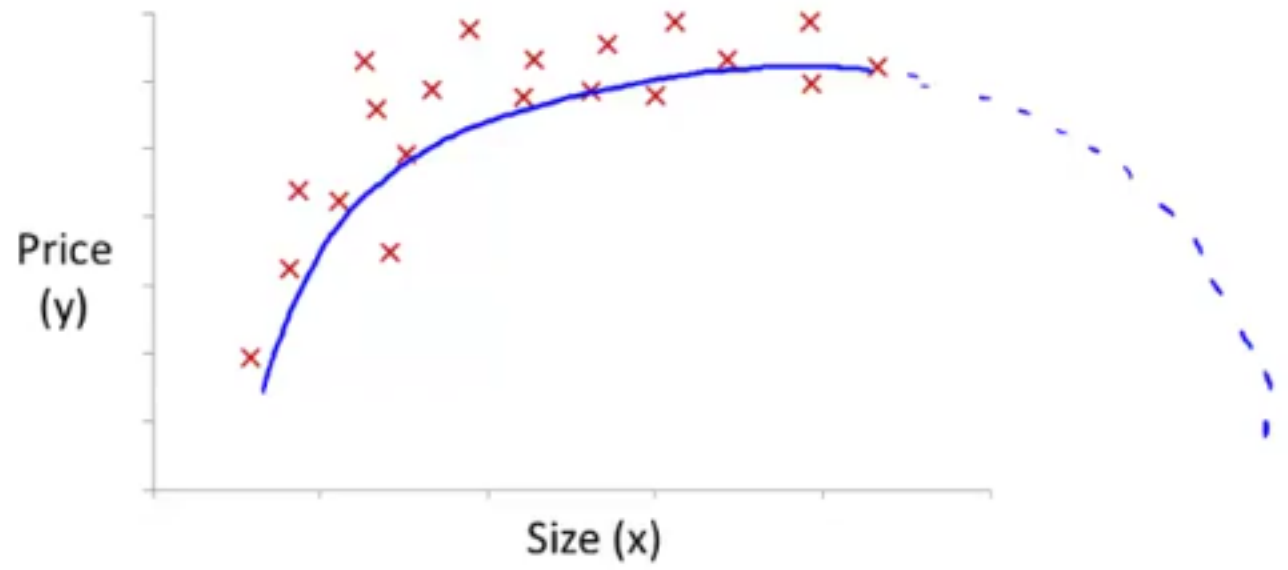


What I actually do is try this range of values. And so on, where this is 0.001. I'll then increase the learning rate threefold to get 0.003. And then this step up, this is another roughly threefold increase from 0.003 to 0.01.

And so these are, roughly, trying out gradient descents with each value I try being about 3x bigger than the previous value. So what I'll do is try a range of values until I've found one value that's too small and made sure that I've found one value that's too large.

And then I'll sort of try to pick the largest possible value, or just something slightly smaller than the largest

Choice of features



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

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

Andrew Ng

Earlier we talked about how a quadratic model like this might not be ideal because, you know, maybe a quadratic model fits the data okay, but the quadratic function goes back down and we really don't want, right, housing prices that go down, to predict that, as the size of housing freezes.

But rather than going to a cubic model there, you have, maybe, other choices of features and there are many possible choices.

But just to give you another example of a reasonable choice, another reasonable choice might be to say

Normal Equation

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

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

Set $A = X^T X$

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

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

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

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

$\min_\theta J(\theta)$

X' X^T

Feature Scaling

$0 \le x_1 \le 1$

$0 \le x_2 \le 1000$

$0 \le x_3 \le 10^{-5}$

If you are using this normal equation method then feature scaling isn't actually necessary and is actually okay if, say, some feature X one is between zero and one, and some feature X two is between ranges from zero to one thousand and some feature x three ranges from zero to ten to the minus five and if you are using the normal equation method this is okay and there is no need to do features scaling, although of course if you are using gradient descent, then, features scaling is still important.

Finally, where should you use the gradient descent and when should you use the normal equation method. Here

m training examples, n features.

Gradient Descent

- Need to choose α.
- Needs many iterations.
- Works well even when n is large.

$n = 10^6$

Normal Equation

- No need to choose α.
- Don't need to iterate.
- Need to compute $(X^T X)^{-1}$ $n \times n$ $O(n^3)$
- Slow if n is very large.

$n = 100$
$n = 1000$
$n = 10000$

Normal Equation

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 $(X^T X)^{-1}$ $n \times n$ $O(n^3)$
	• Slow if n is very large.

$n = 100$
$n = 1000$
$n = 10000$

But, if n is relatively small, then the normal equation might give you a better way to solve the parameters. What does small and large mean? Well, if n is on the order of a hundred, then inverting a hundred-by-hundred matrix is no problem by modern computing standards.

If n is a thousand, I would still use the normal equation method. Inverting a thousand-by-thousand matrix is actually really fast on a modern computer. If n is ten thousand, then I might start to wonder.

Inverting a ten-thousand-by-ten-thousand matrix starts to get