

# Model and Cost Functions

## Model Representation

Our first learning algorithm will be **linear regression**.

In this video, you'll see what the model looks like and more importantly you'll see what the overall process of supervised learning looks like.

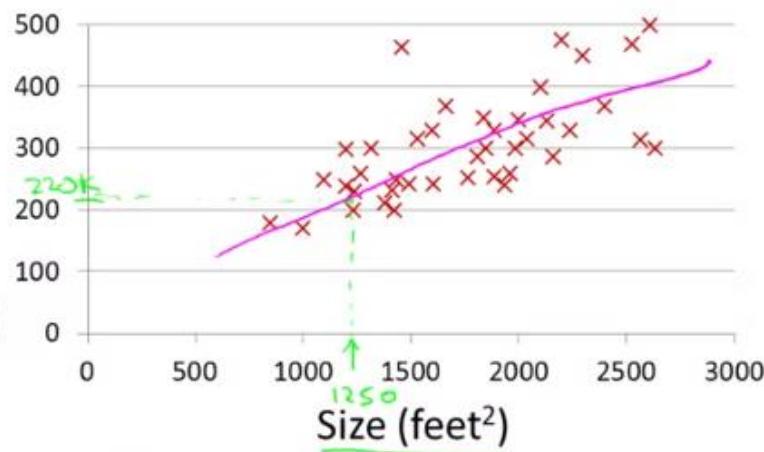
Let's use some motivating example of predicting housing prices. We're going to use a data set of housing prices from the city of Portland, Oregon. And here I'm gonna plot my data set of a number of houses that were different sizes that were sold for a range of different prices. Let's say that given this data set, you have a friend that's trying to sell a house and let's see if friend's house is size of 1250 square feet and you want to tell them how much they might be able to sell the house for. **Well one thing you could do is fit a model**. Maybe fit a straight line to this data.

### Supervised – Regression and Classification

Looks something like that and based on that, maybe you could tell your friend that let's say maybe he can sell the house for around \$220,000. So this is an example of a supervised learning algorithm. And it's **supervised learning** because we're given the, quotes, "right answer" for each of our examples. Namely we're told what was the actual house, what was the actual price of each of the houses in our data set were sold for and moreover, this is an example of a regression problem where the term **regression** refers to the fact that we are predicting a real-valued output namely the price. And just to remind you the other most common type of supervised learning problem is called the **classification problem** where we predict discrete-valued outputs such as if we are looking at cancer tumors and trying to decide if a tumor is malignant or benign. So that's a zero-one valued discrete output.

### Housing Prices (Portland, OR)

Price  
(in 1000s  
of dollars)



Supervised Learning

Given the “right answer” for  
each example in the data.

Regression Problem

Predict real-valued output

Classification: Discrete-valued output

More formally, in supervised learning, we have a data set and this data set is called a training set. So for housing prices example, we have a training set of different housing prices and **our job is to learn from this data how to predict prices of the houses.**

<u>Training set of housing prices (Portland, OR)</u>	<u>Size in feet<sup>2</sup> (x)</u>	<u>Price (\$ in 1000's) (y)</u>
	→ 2104	460
	1416	232
→ 1534		315
852		178
...		...

Notation:

- > **m** = Number of training examples
- > **x**'s = "input" variable / features
- > **y**'s = "output" variable / "target" variable
- $(x, y)$  - one training example
- $(x^{(i)}, y^{(i)})$  - i<sup>th</sup> training example

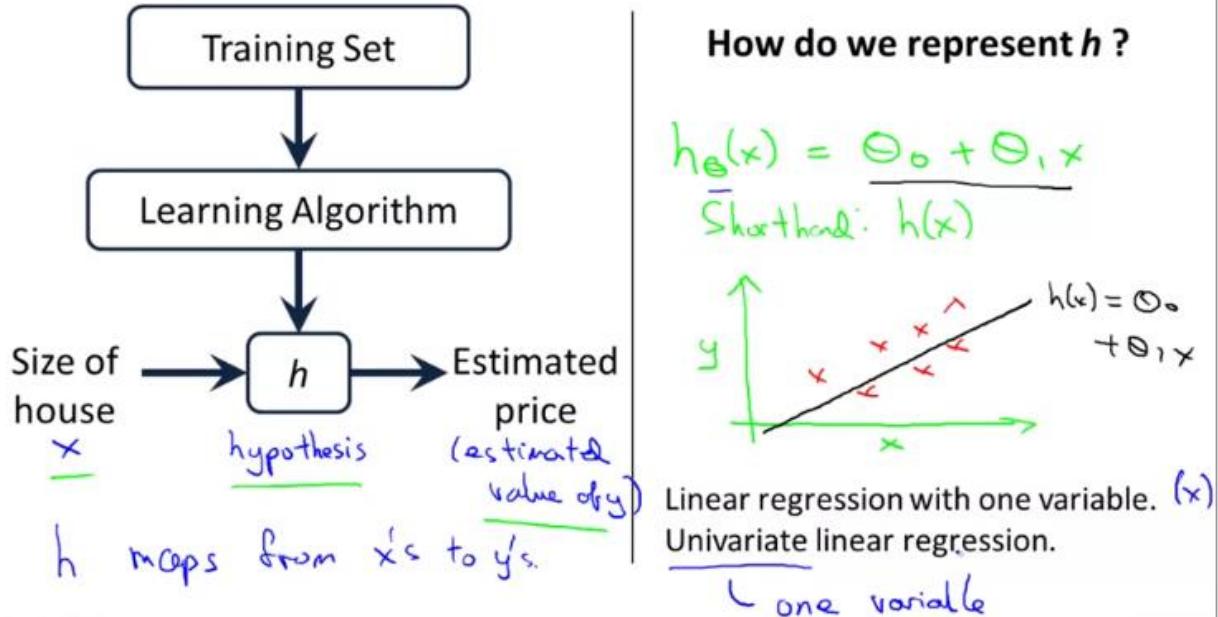
$$\left\{ \begin{array}{l} x^{(1)} = 2104 \\ x^{(2)} = 1416 \\ y^{(1)} = 460 \end{array} \right.$$

Andrew

Let's define some **notation** that we're using throughout this course. We're going to define quite a lot of symbols. It's okay if you don't remember all the symbols right now but as the course progresses it will be useful [inaudible] convenient notation. So I'm gonna use **lower case m** throughout this course to denote the **number of training examples**. So in this data set, if I have, you know, let's say 47 rows in this table. Then I have 47 training examples and m equals 47. Let me use **lowercase x** to denote the **input variables** often also called the **features**. That would be the x is here, it would be the input features. And I'm gonna **use y** to denote my output variables or **the target variable** which I'm **going to predict** and so that's the second column here. [inaudible] notation, I'm going to **use (x, y)** to denote a single training example. So, **a single row in this table corresponds to a single training example** and to refer to a specific training example, I'm going to use this **notation x(i) comma gives me y(i)** And, we're going to use this to refer to the ith training example. So this superscript i over here, this is not exponentiation right? This  $(x(i), y(i))$ , the superscript i in parentheses that's just an index into my training set and refers to the ith row in this table, okay? So this is not x to the power of i, y to the power of i. Instead  $(x(i), y(i))$  just refers to the ith row of this table. So for example,  $x(1)$  refers to the input value for the first training example so that's 2104. That's this x in the first row.  $x(2)$  will be equal to 1416 right? That's the second x and  $y(1)$  will be equal to 460. The first, the y value for my first training example, that's what that (1) refers to.

So as mentioned, occasionally I'll ask you a question to let you check your understanding and a few seconds in this video a multiple-choice question will pop up in the video. When it does, please use your mouse to select what you think is the right answer.

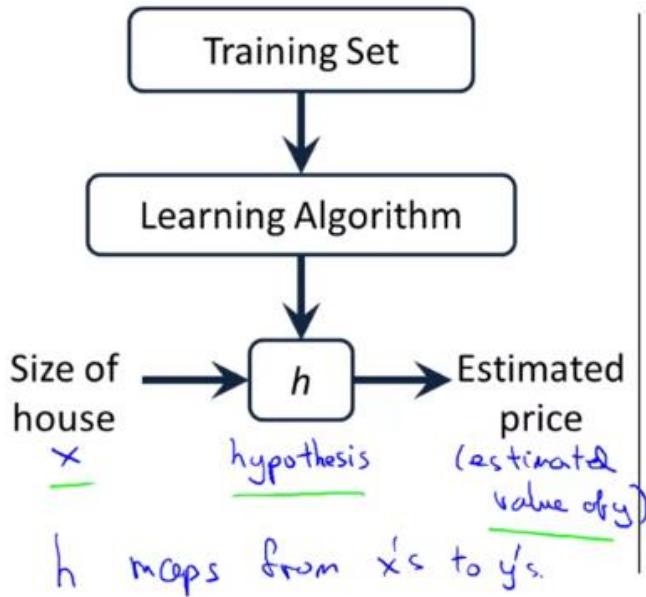
So here's how this supervised learning algorithm works. We saw that with the training set like **our training set of housing prices** and **we feed that to our learning algorithm**. Is the job of a learning algorithm to then output a function which by convention is usually denoted **lowercase h** and **h stands for hypothesis** And what the **job of the hypothesis is**, is, is a function that takes as input the size of a house like maybe the size of the new house your friend's trying to sell so it takes in the value of  $x$  and it tries to output the estimated value of  $y$  for the corresponding house.



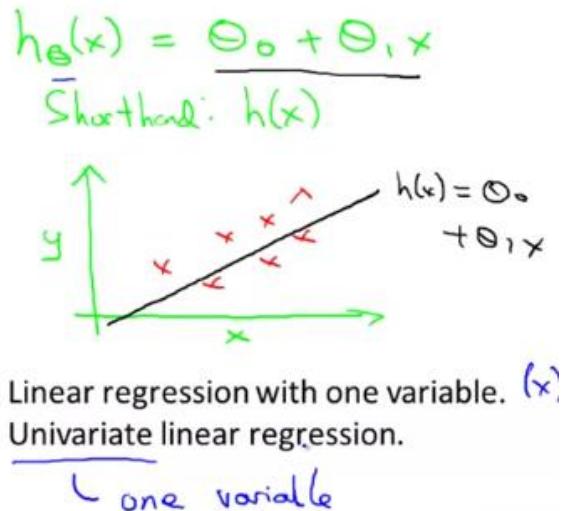
**So  $h$  is a function that maps from  $x$ 's to  $y$ 's.**

People often ask me, you know, why is this function called hypothesis. Some of you may know the meaning of the term hypothesis, from the dictionary or from science or whatever. It turns out that in machine learning, this is a name that was used in the early days of machine learning and it kinda stuck. 'Cause maybe not a great name for this sort of function, for mapping from sizes of houses to the predictions that you know.... I think the term hypothesis, maybe isn't the best possible name for this, but this is the standard terminology that people use in machine learning. So don't worry too much about why people call it that.

When **designing a learning algorithm**, the next thing we need to decide is **how do we represent this hypothesis  $h$** . For this and the next few videos, I'm going to choose **our initial choice**, for representing the hypothesis, will be the following. We're going to represent  $h$  as follows.



### How do we represent $h$ ?



And we will write this as  $h_{\theta}(x)$  equals  $\theta_0 + \theta_1 x$ . And as a shorthand, sometimes instead of writing, you know,  $h$  subscript theta of  $x$ , sometimes there's a shorthand, I'll just write as  $h(x)$ . But more often I'll write it as a subscript theta over there. And plotting this in the pictures, all this means is that, we are going to predict that  $y$  is a linear function of  $x$ . Right, so that's the data set and what this function is doing, is predicting that  $y$  is some straight line function of  $x$ . That's  $h$  of  $x$  equals theta 0 plus theta 1  $x$ , okay?

### And why a linear function?

Well, sometimes we'll want to fit more complicated, perhaps non-linear functions as well. But since this linear case is the simple building block, we will start with this example first of fitting linear functions, and we will build on this to eventually have more complex models, and more complex learning algorithms. Let me also give this particular model a name. This model is called **linear regression** or this, for example, is actually **linear regression with one variable**, with the variable being  $x$ . Predicting all the prices as functions of one variable  $X$ . And another name for this model is **univariate linear regression**. And univariate is just a fancy way of saying one variable. So, that's linear regression. In the next video we'll start to talk about just how we go about implementing this model.

## Cost Function

In this video we'll define something called the **cost function**, this will let us figure out how to fit the best possible straight line to our data.

In linear regression, we have a training set that I showed here remember on **notation m** was the number of training examples, so maybe  $m$  equals 47. And the form of our hypothesis, which we use to make predictions is this linear function. To introduce a little bit more terminology, these

theta zero and theta one, they stabilize what I call the parameters of the model. And what we're going to do in this video is talk about **how to go about choosing these two parameter values, theta 0 and theta 1.**

Training Set	Size in feet <sup>2</sup> (x)	Price (\$) in 1000's (y)
	2104	460
	1416	232
	1534	315
	852	178
	...	...

$\{ m = 47 \}$

Hypothesis: 
$$h_{\theta}(x) = \underline{\theta_0} + \underline{\theta_1}x$$

$\theta_i$ 's: Parameters

How to choose  $\theta_i$ 's ?

With different choices of the parameter's theta 0 and theta 1, we get different hypothesis, different hypothesis functions.

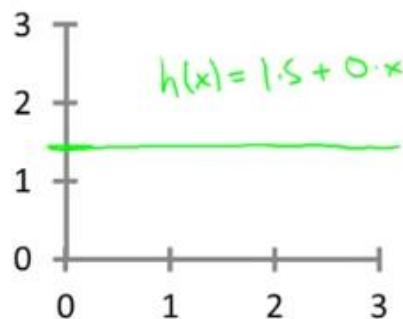
I know some of you will probably be already familiar with what I am going to do on the slide, but just for review, here are a few examples.

If  $\theta_0$  is 1.5 and  $\theta_1$  is 0, then the hypothesis function will look like this. Because your hypothesis function will be  $h$  of  $x$  equals 1.5 plus 0 times  $x$  which is this constant value function which is at 1.5.

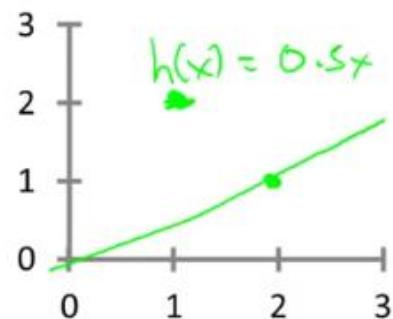
If  $\theta_0 = 0$ ,  $\theta_1 = 0.5$ , then the hypothesis will look like this, and it should pass through this point 2,1 so that you now have  $h(x)$ . Or really  $h$  of  $\theta(x)$ , but sometimes I'll just omit  $\theta$  for brevity. So  $h(x)$  will be equal to just 0.5 times  $x$ , which looks like that.

And finally, if  $\theta_0 = 1$ , and  $\theta_1 = 0.5$ , then we end up with a hypothesis that looks like this. Let's see, it should pass through the two-two point. Like so, and this is my new vector of  $x$ , or my new  $h$  subscript  $\theta$  of  $x$ . Whatever way you remember, I said that this is  $h$  subscript  $\theta$  of  $x$ , but that's a shorthand, sometimes I'll just write this as **h of x**.

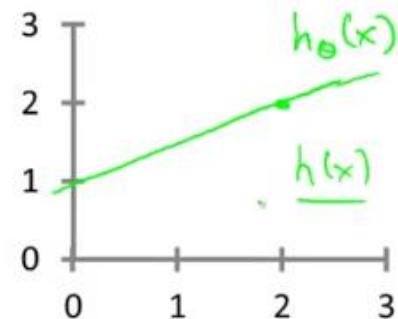
$$\underline{h_{\theta}(x)} = \theta_0 + \theta_1 x$$



$$\begin{aligned} \rightarrow \theta_0 &= 1.5 \\ \rightarrow \theta_1 &= 0 \end{aligned}$$

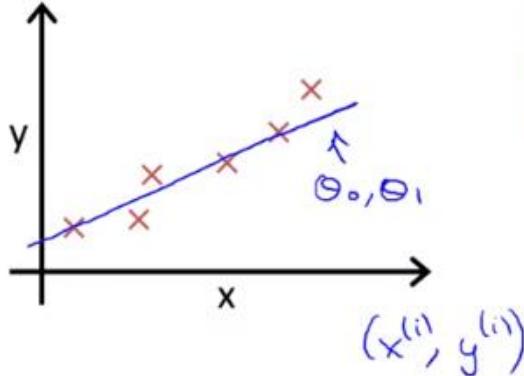


$$\begin{aligned} \rightarrow \theta_0 &= 0 \\ \rightarrow \theta_1 &= 0.5 \end{aligned}$$



$$\begin{aligned} \rightarrow \theta_0 &= 1 \\ \rightarrow \theta_1 &= 0.5 \end{aligned}$$

In linear regression, we have a training set, like maybe the one I've plotted here.



$$\text{minimize}_{\theta_0, \theta_1} \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

$\uparrow$

$h_{\theta}(x^{(i)}) = \underline{\theta_0 + \theta_1 x^{(i)}}$

---


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

Idea: Choose  $\underline{\theta_0}, \underline{\theta_1}$  so that

$\underline{h_{\theta}(x)}$  is close to  $\underline{y}$  for our training examples  $\underline{(x, y)}$

$\underline{x}, \underline{y}$

$\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$

$\underbrace{J(\theta_0, \theta_1)}$  Cost function

Squared error function

Andrew Ng

What we want to do, is come up with values for the parameters theta zero and theta one so that the straight line we get out of this, corresponds to a straight line that somehow fits the data well, like maybe that line over there.

So, how do we come up with values, theta0, theta1, that corresponds to a good fit to the data? The idea is we get to choose our parameters theta0, theta1 so that  $h(x)$ , meaning the value we predict on input  $x$  is at least close to the values  $y$  for the examples in our training set, for our training examples.

So in our training set, we've given a number of examples where we know X decides the house and we know the actual price house was sold for. So, let's try to choose values for the parameters so that, at least in the training set, given the X in the training set we make reason of the active predictions for the Y values.

Let's formalize this. So in linear regression, **what we're going to do is**, I'm **going to want to solve a minimization problem**. So I'll write minimize over theta0 theta1. And I want this to be small, right? I want the difference between  $h(x)$  and  $y$  to be small. And one thing I might do is try to **minimize the square difference** between the output of the hypothesis and the actual price of a house.

Okay. So let's find some details.

You remember that I was using the notation  $(x(i), y(i))$  to represent the  $i$ th training example. So what I want really is to sum over my training set, something  $i = 1$  to  $m$ , of the square difference between, this is the prediction of my hypothesis when it is input to size of house number  $i$ . Right? Minus the actual price that house number  $i$  was sold for, and I want to minimize the sum of my training set, **sum from  $i$  equals one through  $m$** , of the difference of this squared error, the square difference between the predicted price of a house, and the price that it was actually sold for. And just remind you of notation,  **$m$  here was the size of my training set** right? So my  $m$  there is my number of training examples.

Right that hash sign is the abbreviation for number of training examples, okay?

And to make some of our, make the math a little bit easier, I'm going to actually look at we are 1 over  $m$  times that so let's try to minimize my average minimize one over  $2m$ . Putting the 2 at the constant one half in front, it may just sound the math probably easier so minimizing one-half of something, right, should give you the same values of the process, theta0 theta1, as minimizing that function.

And just to be sure, this equation is clear, right? This expression in here,  $h$  subscript theta( $x$ ), this is our usual, right? That is equal to this plus theta one  $x_i$ . And this notation, minimize over theta 0 theta1, this means you'll find me the values of theta0 and theta1 that causes this expression to be minimized and this expression depends on theta0 and theta1, okay?

So just a recap.

We're closing this problem as, **find me the values of theta0 and theta1** so that the average, the 1 over the  $2m$ , times the sum of square errors between my predictions on the training set minus the actual values of the houses on the training set is minimized.

So **this is going to be my overall objective function for linear regression**.

And just to rewrite this out a little bit more cleanly, what I'm going to do is, by convention we usually define a **cost function**, which is going to be exactly this, that formula I have up here. And what I want to do is minimize over theta0 and theta1.

My function **j(theta0, theta1)**.

Just write this out. This is my cost function. So, this cost function is also called the **squared error function**.

When sometimes called the squared error cost function and it turns out that why do we take the squares of the errors. It turns out that these squared error cost function is a reasonable choice and works well for problems for most regression programs. There are other cost functions that will work pretty well. But the square cost function is probably the most commonly used one for regression problems.

Later in this class we'll talk about alternative cost functions as well, but this choice that we just had should be a pretty reasonable thing to try for most linear regression problems. Okay. So that's the cost function.

So far we've just seen a mathematical definition of this cost function. In case this function  $J$  of  $\theta_0, \theta_1$ . In case this function seems a little bit abstract, and you still don't have a good sense of what it's doing, in the next video, in the next couple videos, I'm actually going to go a little bit deeper into **what the cost function "J" is doing** and try to give you better intuition about **what it is computing and why we want to use it...**

## Cost Function - Intuition I

In the previous video, we gave the mathematical definition of the cost function. In this video, let's look at some examples, to get back to intuition about what the cost function is doing, and why we want to use it.

To recap, here's what we had last time.

**Hypothesis:**

$$\underline{h_{\theta}(x) = \theta_0 + \theta_1 x}$$

**Parameters:**

$$\underline{\theta_0, \theta_1}$$

**Cost Function:**

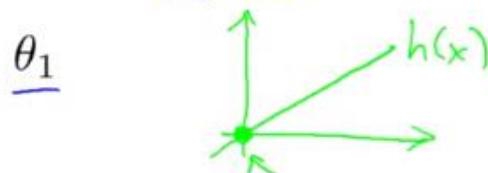
$$\rightarrow J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

**Goal:** minimize  $J(\theta_0, \theta_1)$

Simplified

$$h_{\theta}(x) = \underline{\theta_1 x}$$

$$\underline{\theta_0 = 0}$$



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

$$\min_{\theta_1} \underline{J(\theta_1)}$$

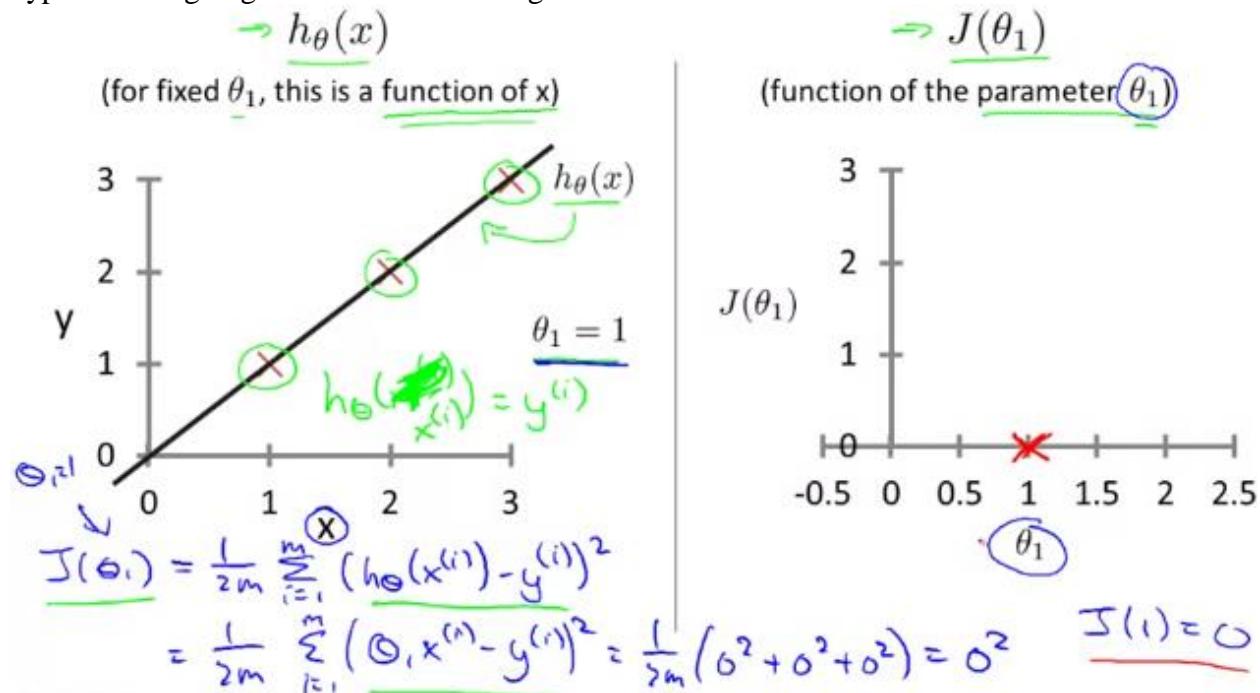
We want to fit a straight line to our data, so we had this formed as a hypothesis with these parameters theta zero and theta one, and with different choices of the parameters we end up with different straight line fits. So the data which are fit like so, and there's a cost function, and that was our optimization objective.

For this video, in order to better visualize the cost function  $J$ , I'm going to work with a simplified hypothesis function, like that shown on the right. So I'm gonna use my simplified hypothesis, which is just  $\theta_1$  times  $X$ . We can, if you want, think of this as setting the parameter  $\theta_0$  equal to 0. So I have only one parameter  $\theta_1$  and my cost function is similar to before except that now  $H$  of  $X$  that is now equal to just  $\theta_1$  times  $X$ . And I have only one parameter  $\theta_1$  and so my optimization objective is to minimize  $J$  of  $\theta_1$ . In pictures what this means is that if  $\theta_0$  equals zero that corresponds to choosing only hypothesis functions that pass through the origin, that pass through the point  $(0, 0)$ . Using this simplified definition of a hypothesizing cost function let's try to understand the cost function concept better.

It turns out that two key functions we want to understand.

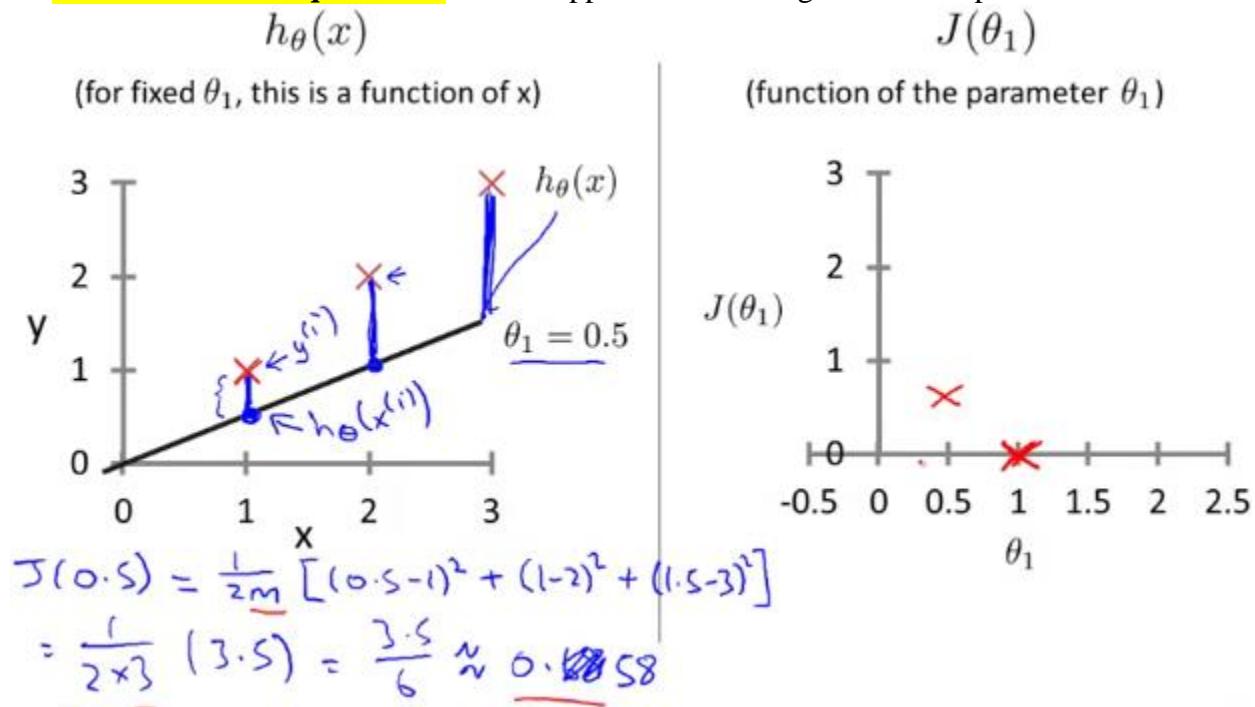
The first is the hypothesis function, and the second is a cost function. So, notice that the hypothesis, right,  $H$  of  $X$ . For a fixed value of  $\theta_1$ , this is a function of  $X$ . So the **hypothesis is a function of  $x$** , what is the size of the house  $X$ . In contrast, the **cost function,  $J$ , that's a function of the parameter,  $\theta_1$** , which controls the slope of the straight line.

Let's plot these functions and try to understand them both better. Let's start with the hypothesis. On the left, let's say here's my training set with three points at  $(1, 1)$ ,  $(2, 2)$ , and  $(3, 3)$ . Let's pick a value  $\theta_1$ , so **when  $\theta_1$  equals one**, and if that's my choice for  $\theta_1$ , then my hypothesis is going to look like this straight line over here.



And I'm gonna point out, when I'm plotting my hypothesis function. X-axis, my horizontal axis is labeled X, is labeled you know, size of the house over here. Now, of temporary, set theta one equals one, what I want to do is figure out what is j of theta one, when theta one equals one. So let's go ahead and compute what the cost function has for the value one. Well, as usual, my cost function is defined as follows, right? Some from, some of 'em are training sets of this usual squared error term. And, this is therefore equal to this. Of theta one x I minus y I and if you simplify this turns out to be. That. Zero Squared to zero squared to zero squared which is of course, just equal to zero. Now, inside the cost function. It turns out each of these terms here is equal to zero. Because for the specific training set I have or my 3 training examples are (1, 1), (2, 2), (3, 3). If theta one is equal to one. Then h of x. H of x i. Is equal to y I exactly, let me write this better. Right? And so, h of x minus y, each of these terms is equal to zero, which is why I find that j of one is equal to zero. So, we now know that j of one Is equal to zero. Let's plot that. What I'm gonna do on the right is plot my cost function j. And notice, because my cost function is a function of my parameter theta one, when I plot my cost function, the horizontal axis is now labeled with theta one. So I have j of one equals zero so let's go ahead and plot that. End up with. An X over there. Now let's look at some other examples. Theta-1 can take on a range of different values. Right? So theta-1 can take on the negative values, zero, positive values.

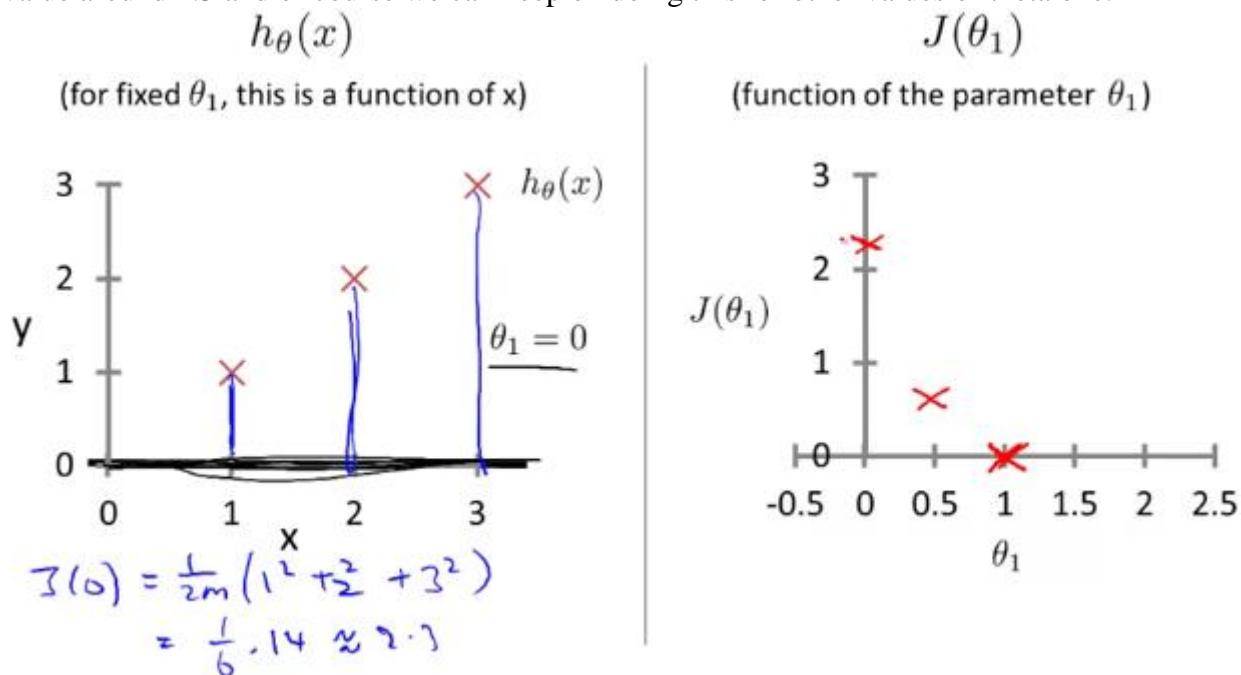
So **what if theta-1 is equal to 0.5**. What happens then? Let's go ahead and plot that.



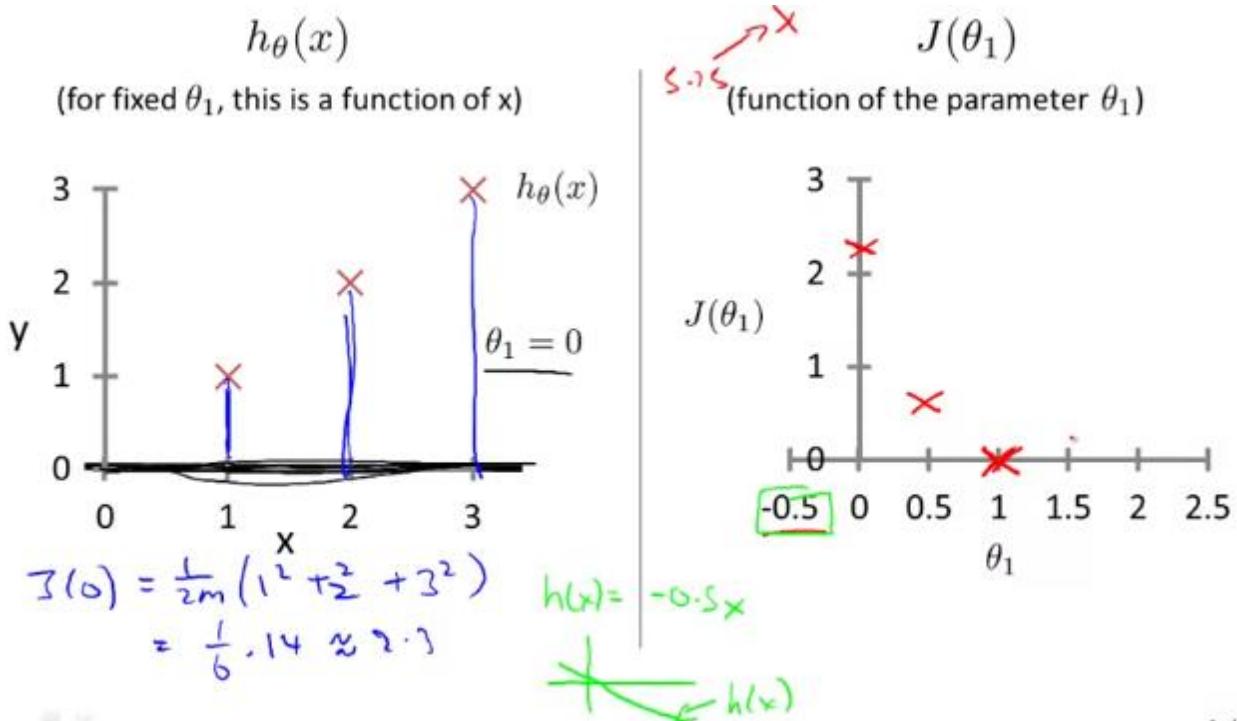
I'm now going to set theta-1 equals 0.5, and in that case my hypothesis now looks like this. As a line with slope equals to 0.5, and, lets compute J, of 0.5. So that is going to be one over 2M of, my usual cost function. It turns out that the cost function is going to be the sum of square values of the height of this line. Plus the sum of square of the height of that line, plus the sum of square of the height of that line, right? ?Cause just this vertical distance, that's the difference between, you know, Y. I. and the predicted value, H of XI, right? So the first example is going to be 0.5 minus one squared. Because my hypothesis predicted 0.5. Whereas, the actual value was one.

For my second example, I get, one minus two squared, because my hypothesis predicted one, but the actual housing price was two. And then finally, plus. 1.5 minus three squared. And so that's equal to one over two times three. Because, M when trading set size, right, have three training examples. In that, that's times simplifying for the parentheses it's 3.5. So that's 3.5 over six which is about 0.68. So now we know that  $J$  of 0.5 is about 0.68.[Should be 0.58] Lets go and plot that. Oh excuse me, math error, it's actually 0.58. So we plot that which is maybe about over there. Okay?

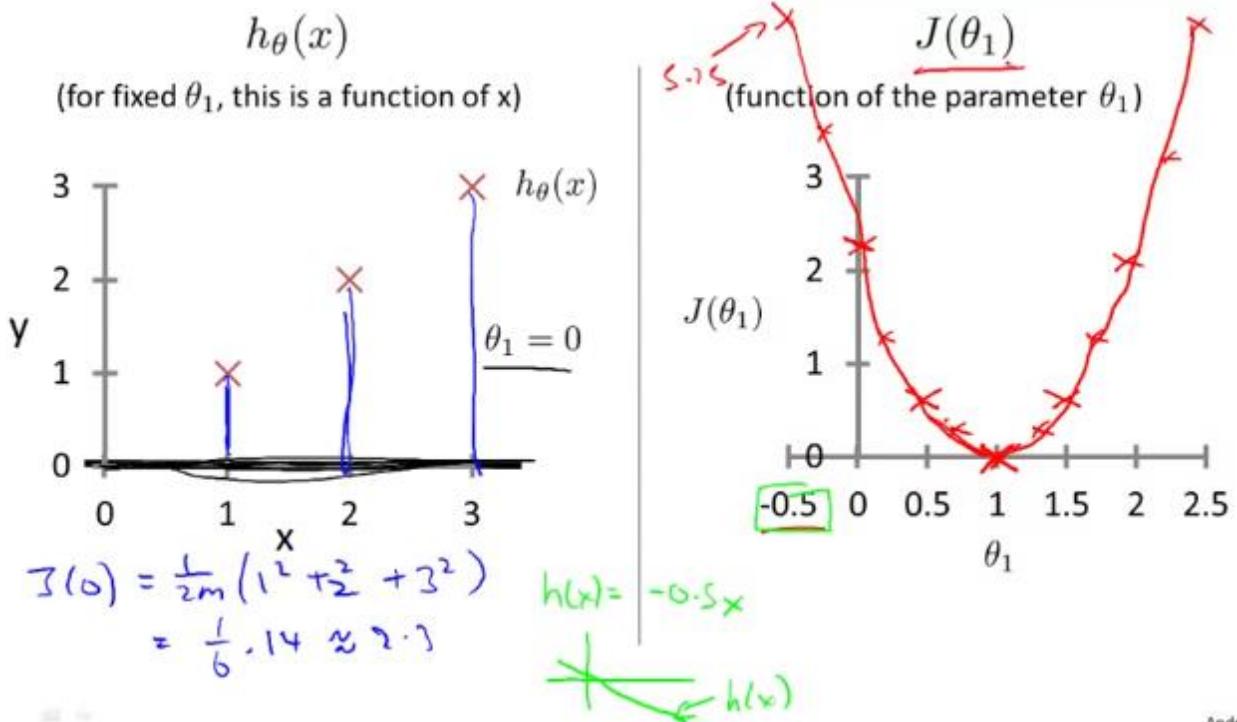
Now, let's do one more. How about if theta one is equal to zero, what is  $J$  of zero equal to? It turns out that if theta one is equal to zero, then  $H$  of  $X$  is just equal to, you know, this flat line, right, that just goes horizontally like this. And so, measuring the errors. We have that  $J$  of zero is equal to one over two  $M$ , times one squared plus two squared plus three squared, which is, One six times fourteen which is about 2.3. So let's go ahead and plot as well. So it ends up with a value around 2.3 and of course we can keep on doing this for other values of theta one.



It turns out that you can have you know **negative values of theta one** as well so if theta one is negative then  $h$  of  $x$  would be equal to say minus 0.5 times  $x$  then theta one is minus 0.5 and so that corresponds to a hypothesis with a slope of negative 0.5. And you can actually keep on computing these errors. This turns out to be, you know, for 0.5, it turns out to have really high error. It works out to be something, like, 5.25.



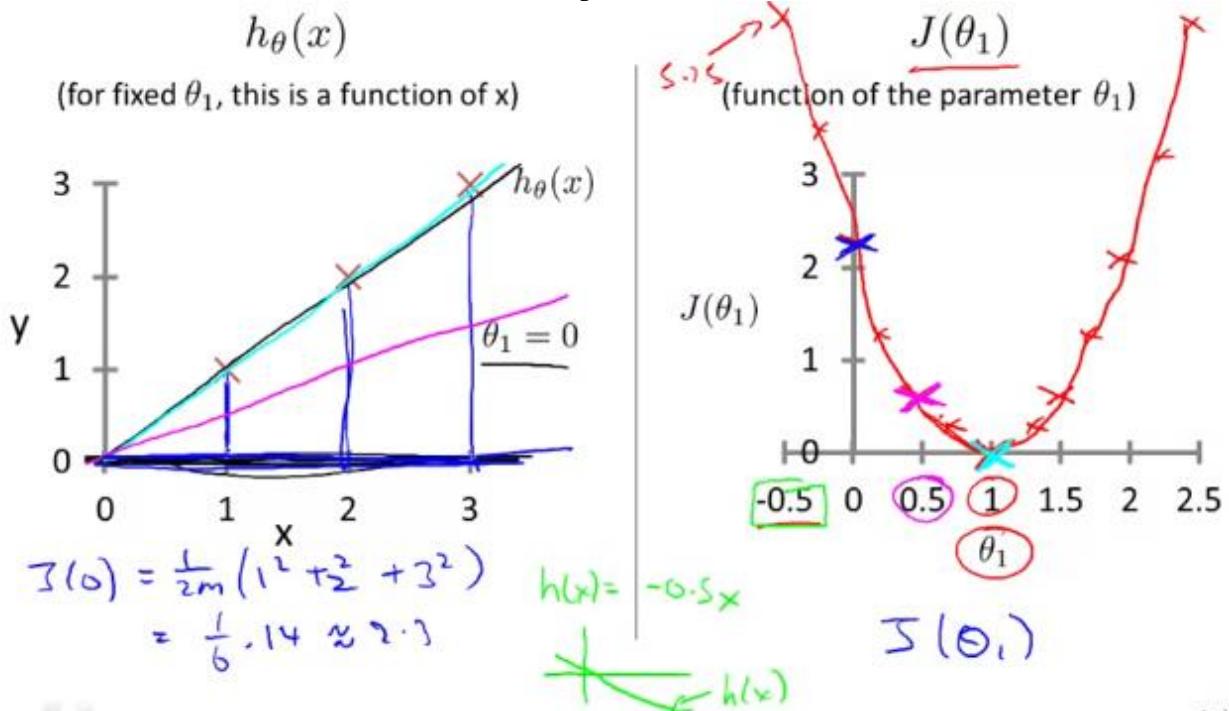
And so on, and the **different values of theta one**, you can compute these things, right?



And it turns out that you, your computed range of values, you get something like that. And by computing the range of values, you can actually slowly create out. What this function  $J$  of Theta looks like, and that's what  $J$  of Theta is. To recap, for each value of theta one, right? **Each value of theta one corresponds to a different hypothesis, or to a different straight line fit on the left.**

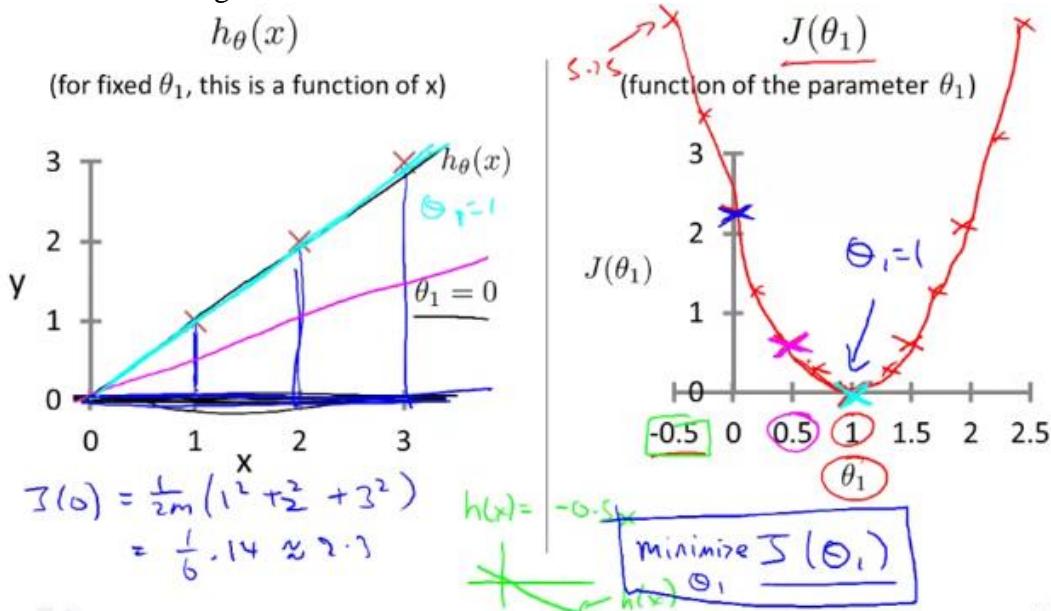
And for each value of theta one, we could then derive a **different value of  $J$  of theta one**. And for example, you know, theta one=1, corresponded to this straight line straight through the data.

Whereas theta one=0.5. And this point shown in magenta corresponded to maybe that line, and theta one=0 which is shown in blue that corresponds to this horizontal line.



Right, so for each value of theta one we wound up with a different value of  $J$  of theta one and we could then use this to trace out this plot on the right.

Now you remember, the **optimization objective for our learning algorithm is we want to choose the value of theta one that minimizes  $J$  of theta one**. Right? This was our objective function for the linear regression.



Well, looking at this curve, the value that minimizes  $J$  of theta one is, you know, theta one equals to one. And low and behold, that is indeed the best possible straight line fit through our data, by

setting theta one equals one. And just, for this particular training set, we actually end up fitting it perfectly. And that's why minimizing  $J$  of theta one corresponds to finding a straight line that fits the data well.

So, to wrap up. In this video, we looked up some plots to understand the cost function. To do so, we simplify the algorithm. So that it only had one parameter theta one. And we set the parameter theta zero to be only zero. In the next video. We'll go back to the original problem formulation and look at some visualizations involving both theta zero and theta one. That is without setting theta zero to zero. And hopefully that will give you, an even better sense of what the cost function  $J$  is doing in the original linear regression formulation.

## Cost Function - Intuition II

In this video, let's delve deeper and get even better intuition about what the cost function is doing.

This video assumes that you're familiar with contour plots. If you are not familiar with contour plots or contour figures some of the illustrations in this video may or may not make sense to you but is okay and if you end up skipping this video or some of it does not quite make sense because you haven't seen contour plots before. That's okay and you will still understand the rest of this course without those parts of this.

Here's our problem formulation as usual, with the hypothesis parameters, cost function, and our optimization objective. Unlike before, unlike the last video, I'm **going to keep both of my parameters, theta zero, and theta one**, as we generate our visualizations for the cost function. So, same as last time, **we want to understand the hypothesis  $H$  and the cost function  $J$** .

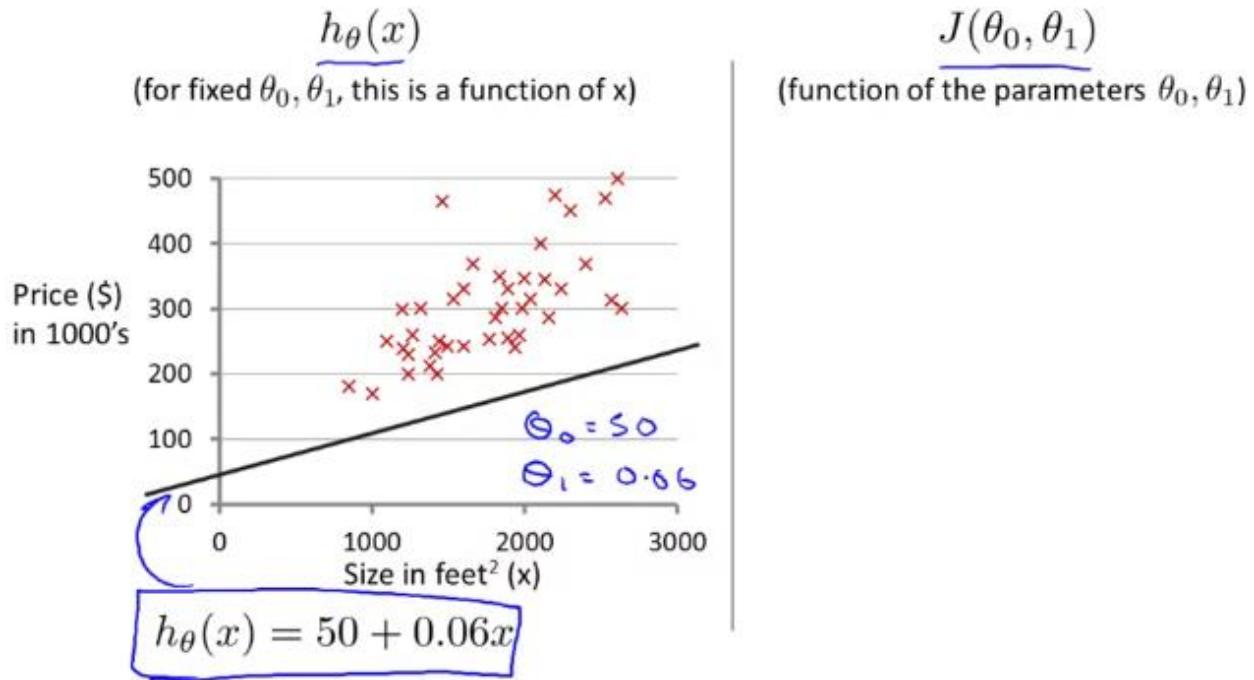
**Hypothesis:** 
$$h_{\theta}(x) = \theta_0 + \theta_1 x$$

**Parameters:**  $\theta_0, \theta_1$

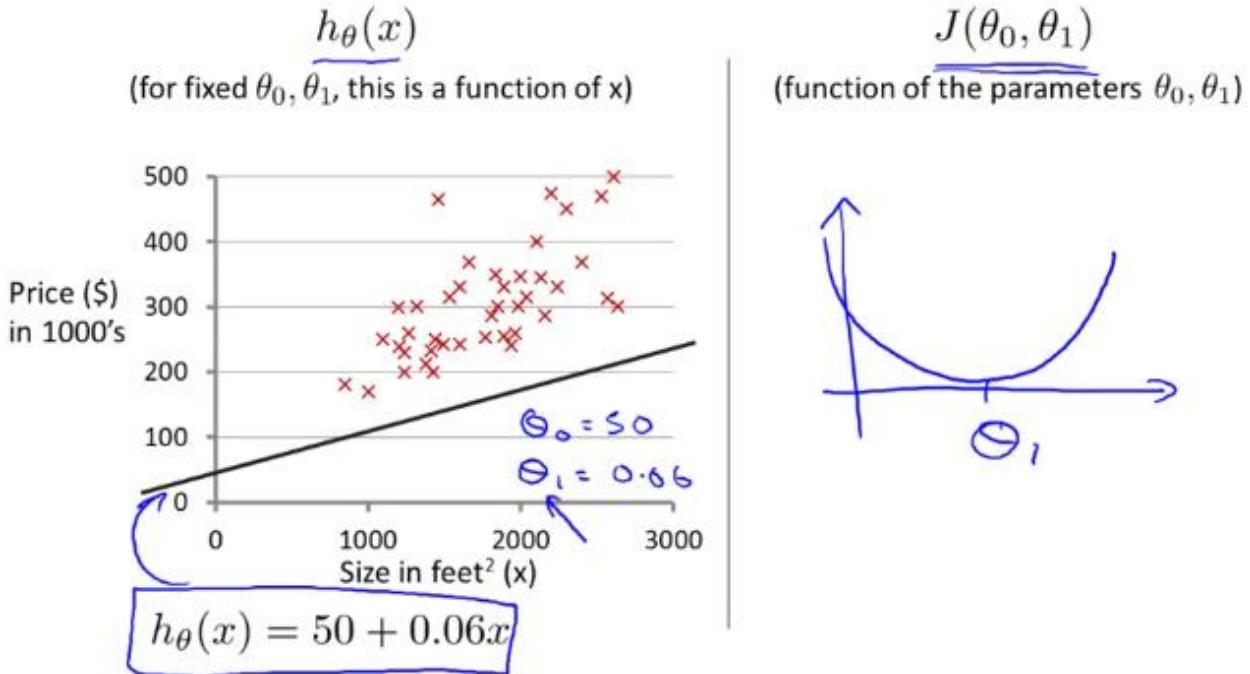
**Cost Function:** 
$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

**Goal:** 
$$\underset{\theta_0, \theta_1}{\text{minimize}} J(\theta_0, \theta_1)$$

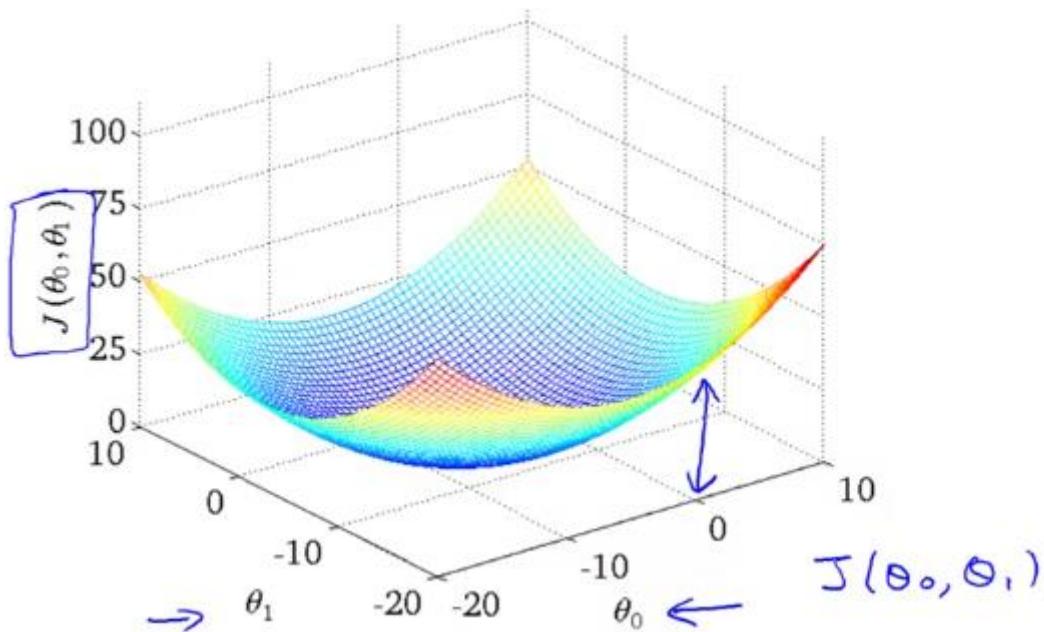
So, here's my training set of housing prices and let's make some hypothesis. You know, like that one, this is not a particularly good hypothesis. But, if I set theta zero=50 and theta one=0.06, then I end up with this hypothesis down here and that corresponds to that straight line.



Now given these value of theta zero and theta one, we want to plot the corresponding, you know, cost function on the right. What we did last time was, right, when we only had theta one. In other words, drawing plots that look like this as a function of theta one.

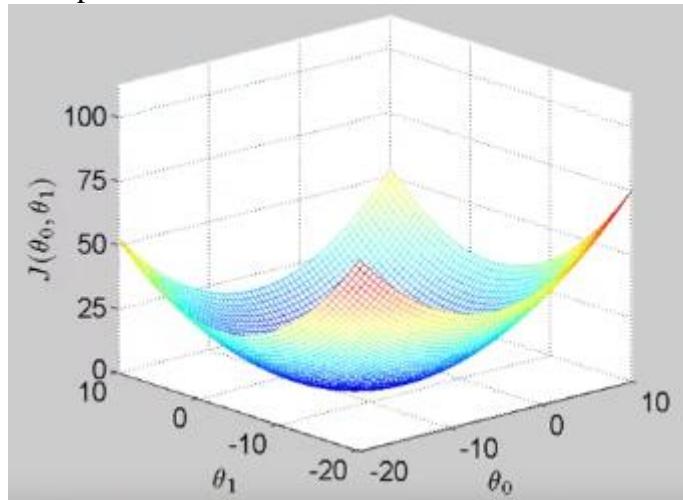


But now we have two parameters, theta zero, and theta one, and so the plot gets a little more complicated. It turns out that when we have only one parameter, that the parts we drew had this sort of bow shaped function. Now, when we have two parameters, it turns out the cost function also has a similar sort of bow shape.

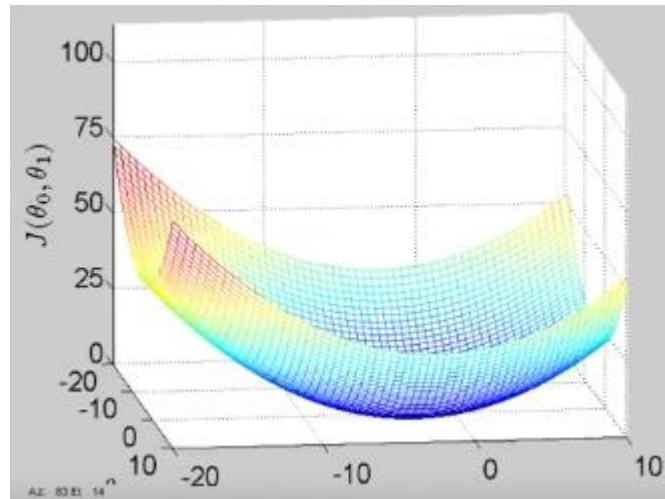


And, in fact, depending on your training set, you might get a cost function that maybe looks something like this. So, this is a 3-D surface plot, where the axes are labeled theta zero and theta one. So as you vary theta zero and theta one, the two parameters, you get different values of the cost function  $J$  (theta zero, theta one) and the height of this surface above a particular point of theta zero, theta one. Right, that's, that's the vertical axis. The height of the surface of the points indicates the value of  $J$  of theta zero,  $J$  of theta one. And you can see it sort of has this bow like shape.

Let me show you the same plot in 3D.

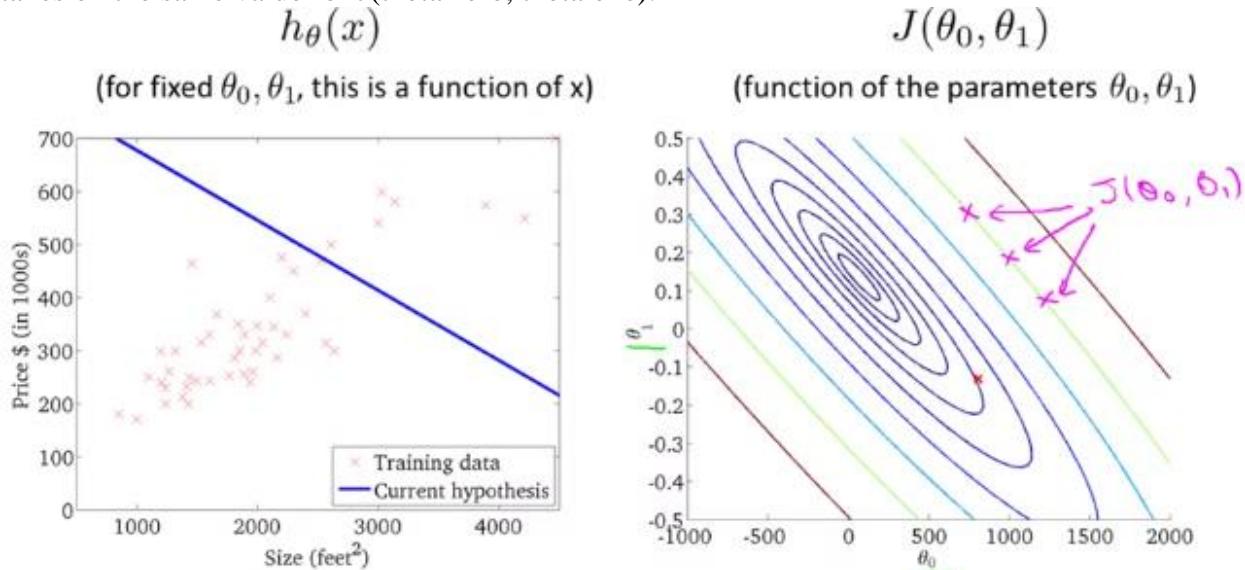


So here's the same figure in 3D, horizontal axis theta one and vertical axis  $J(\theta_0, \theta_1)$ , and if I rotate this plot around. You kinda of a get a sense, I hope, of this bowl shaped surface as that's what the cost function  $J$  looks like.



Now for the purpose of illustration in the rest of this video I'm not actually going to use these sort of 3D surfaces to show you the cost function  $J$ , instead **I'm going to use contour plots**. Or what I also call contour figures. I guess they mean the same thing. To show you these surfaces.

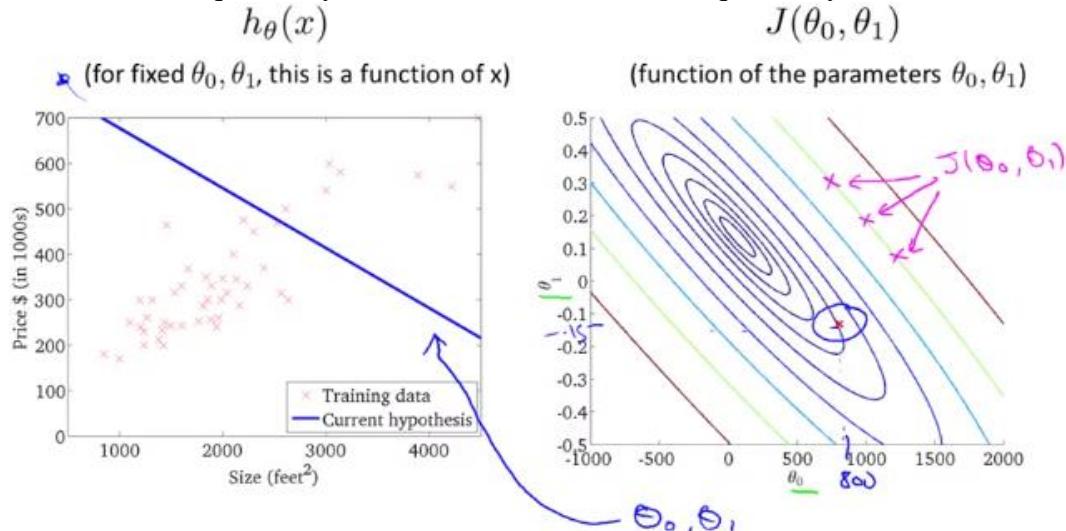
So here's an **example of a contour figure**, shown on the right, where the axis are theta zero and theta one. And what each of these ellipses, what each of these ellipsis shows is a set of points that takes on the same value for  $J(\theta_0, \theta_1)$ .



So concretely, for example this, you'll take that point and that point and that point. All three of these points that I just drew in magenta, they have the same value for  $J(\theta_0, \theta_1)$ . Okay. Where, right, these, this is the theta zero, theta one axis but those three have the same Value for  $J(\theta_0, \theta_1)$  and if you haven't seen contour plots much before think of, imagine if you will. A bow shaped function that's coming out of my screen. So that the minimum, so the bottom of the bow is this point right there, right? This middle, the middle of these concentric ellipses. And imagine a bow shape that sort of grows out of my screen like this, so that each of these ellipses, you know, has the same height above my screen. And the minimum

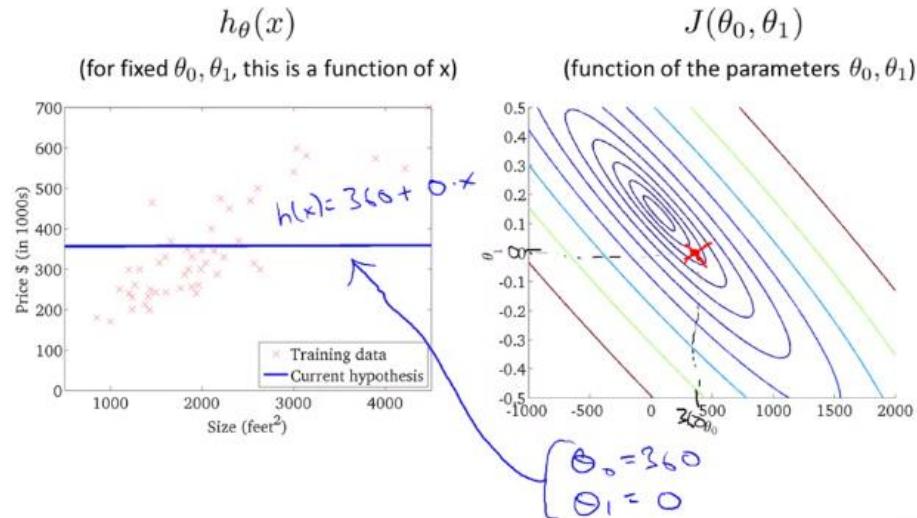
with the bow, right, is right down there. And so the contour figures is a, is way to, is maybe a more convenient way to visualize my function  $J$ .

So, let's look at some examples. Over here, I have a particular point, right? And so this is, with, you know, theta zero equals maybe about 800, and theta one equals maybe a -0.15 .

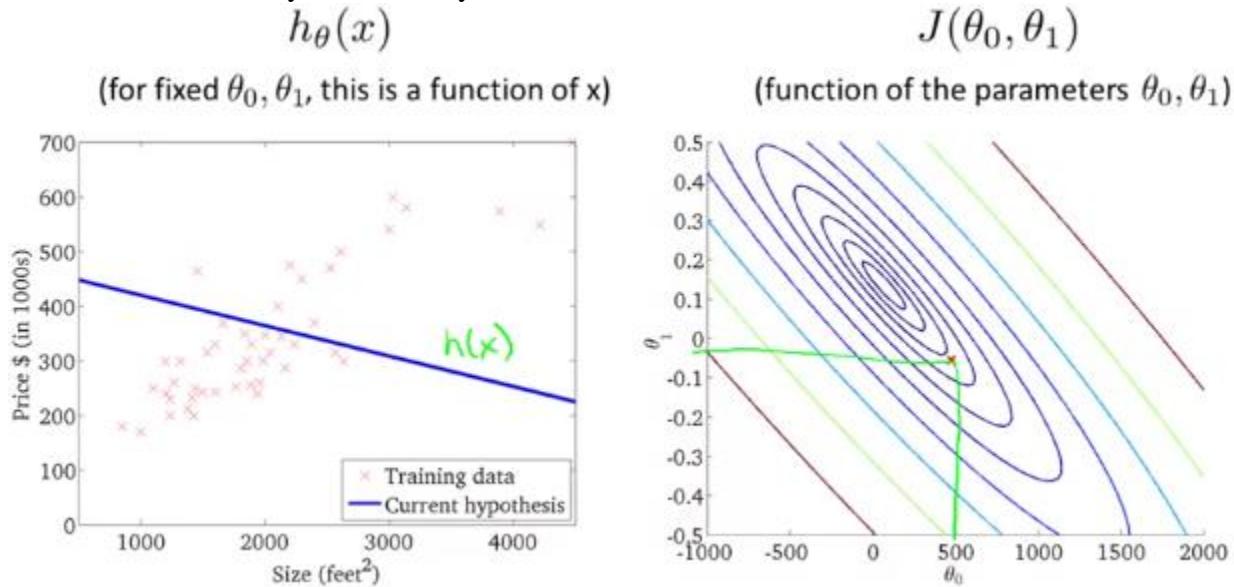


And so this point, right, this point in red corresponds to one set of pair values of theta zero, theta one and the corresponding, in fact, to that hypothesis, right, **theta zero is about 800**, that is, where it intersects the vertical axis is around 800, and this is **slope of about -0.15**. Now this line is really **not such a good fit to the data**, right. This hypothesis,  $h(x)$ , with these values of theta zero, theta one, it's really not such a good fit to the data. And so you find that, it's **cost is a value that's out here that's you know pretty far from the minimum** right it's pretty far this is a pretty high cost because this is just not that good a fit to the data.

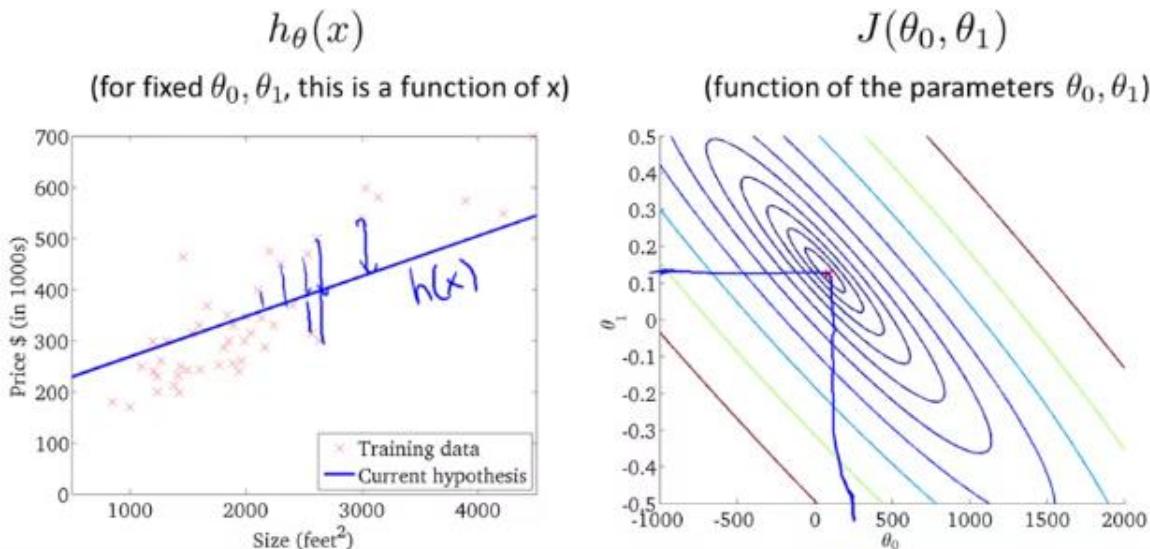
Let's look at some more examples. Now here's a different hypothesis that's you know still not a great fit for the data but may be slightly better so here right that's my point that those are my parameters theta zero theta one and so my theta zero value. Right? That's bout 360 and my value for theta one is equal to zero. So, you know, let's break it out. Let's take **theta zero equals 360** **theta one equals zero**. And this pair of parameters corresponds to that hypothesis, corresponds to flat line, that is,  $h(x)$  equals 360 plus zero times  $x$ . So that's the hypothesis. And this hypothesis again has some cost, and that cost is, you know, plotted as the height of the  $J$  function at that point.



Let's look at just a couple of examples. Here's one more, you know, at this value of theta zero, and at that value of theta one, we end up with this hypothesis,  $h(x)$  and again, not a great fit to the data, and is actually further away from the minimum.



Last example, this is actually **not quite at the minimum**, but it's pretty close to the minimum.



So this is not such a bad fit to the data, where, for a particular value, of, theta zero. Which, one of them has value, as in for a particular value for theta one. We get a particular  $h(x)$ . And this is, this is not quite at the minimum, but it's pretty close. And so the sum of squares errors is sum of squares distances between my, training samples and my hypothesis. Really, that's a sum of square distances, right? Of all of these errors. This is pretty close to the minimum even though it's not quite the minimum.

So with these figures I hope that gives you a better understanding of what values of the cost function  $J$ , how they are and how that corresponds to different hypothesis and so as how better hypotheses may corresponds to points that are closer to the minimum of this cost function  $J$ . Now of course what we really want is an efficient algorithm, right, a efficient piece of software for automatically finding the value of theta zero and theta one, that minimizes the cost function  $J$ , right?

And what we, what we don't wanna do is to, you know, how to write software, to plot out this point, and then try to manually read off the numbers, that this is not a good way to do it. And, in fact, we'll see it later, that when we look at more complicated examples, we'll have high dimensional figures with more parameters, that, it turns out, we'll see in a few, we'll see later in this course, examples where this figure, you know, cannot really be plotted, and this becomes much harder to visualize. And so, what we want is to have software to find the value of theta zero, theta one that minimizes this function and in the next video we start to talk about an algorithm for automatically finding that value of theta zero and theta one that minimizes the cost function  $J$ .

## Parameter Learning

### Gradient Descent

We previously defined the cost function  $J$ .

In this video, I want to tell you about an **algorithm called gradient descent for minimizing the cost function J.**

It turns out gradient descent is a more general algorithm, and is used not only in linear regression. It's actually used all over the place in machine learning. And later in the class, we'll use gradient descent to minimize other functions as well, not just the cost function J for the linear regression.

So in this video, we'll talk about **gradient descent for minimizing some arbitrary function J** and then in later videos, we'll take this algorithm and apply it specifically to the cost function J that we have defined for linear regression. So here's the problem setup.

$$\text{Have some function } \underline{J(\theta_0, \theta_1)} \quad J(\theta_0, \theta_1, \theta_2, \dots, \theta_n)$$

$$\text{Want } \min_{\theta_0, \theta_1} \underline{J(\theta_0, \theta_1)} \quad \min_{\theta_0, \dots, \theta_n} \underline{J(\theta_0, \dots, \theta_n)}$$

### Outline:

- Start with some  $\theta_0, \theta_1$
  - Keep changing  $\theta_0, \theta_1$  to reduce  $J(\theta_0, \theta_1)$
- until we hopefully end up at a minimum

Going to assume that we have some function  $J(\theta_0, \theta_1)$  maybe it's the cost function from linear regression, maybe it's some other function we wanna minimize. And we want to come up with an algorithm for minimizing that as a function of  $J(\theta_0, \theta_1)$ . Just as an aside it turns out that gradient descent actually applies to more general functions.

So imagine, if you have a function that's a function of  $J$ , as  $\theta_0, \theta_1, \theta_2$ , up to say some  $\theta_n$ , and you want to minimize  $\theta_0$ . You minimize over  $\theta_0$  up to  $\theta_n$  of this  $J$  of  $\theta_0$  up to  $\theta_n$ . And it turns out gradient descent is an algorithm for solving this more general problem. But for the sake of brevity, for the sake of succinctness of notation, I'm just going to pretend I have only two parameters throughout the rest of this video.

Here's the idea for gradient descent. What we're going to do is we're going to start off with some initial guesses for  $\theta_0$  and  $\theta_1$ . Doesn't really matter what they are, but a common choice would be we set  $\theta_0$  to 0, and set  $\theta_1$  to 0, just initialize them to 0. What we're going to do in gradient descent is we'll keep changing  $\theta_0$  and  $\theta_1$  a little bit to try to reduce  $J(\theta_0, \theta_1)$ , until hopefully, we wind at a minimum, or maybe at a local minimum.

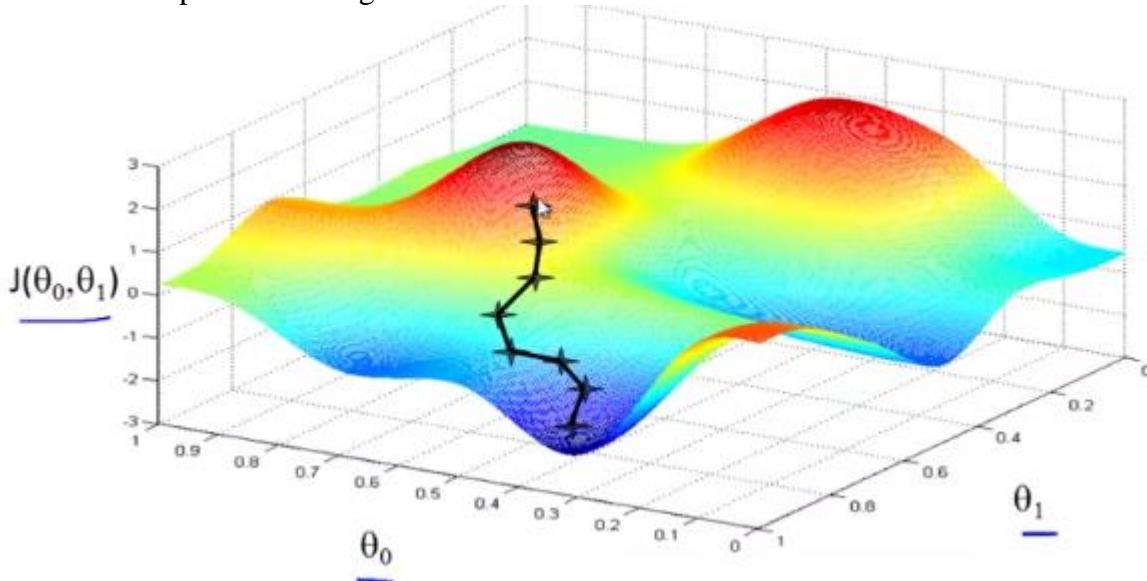
Have some function  $J(\theta_0, \theta_1)$   $\underline{J(\theta_0, \theta_1, \theta_2, \dots, \theta_n)}$

Want  $\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$   $\underline{\min_{\theta_0, \dots, \theta_n} J(\theta_0, \dots, \theta_n)}$

## Outline:

- Start with some  $\underline{\theta_0, \theta_1}$  (say  $\theta_0 = 0, \theta_1 = 0$ )
- Keep changing  $\underline{\theta_0, \theta_1}$  to reduce  $\underline{J(\theta_0, \theta_1)}$   
until we hopefully end up at a minimum

So let's see in pictures what gradient descent does.

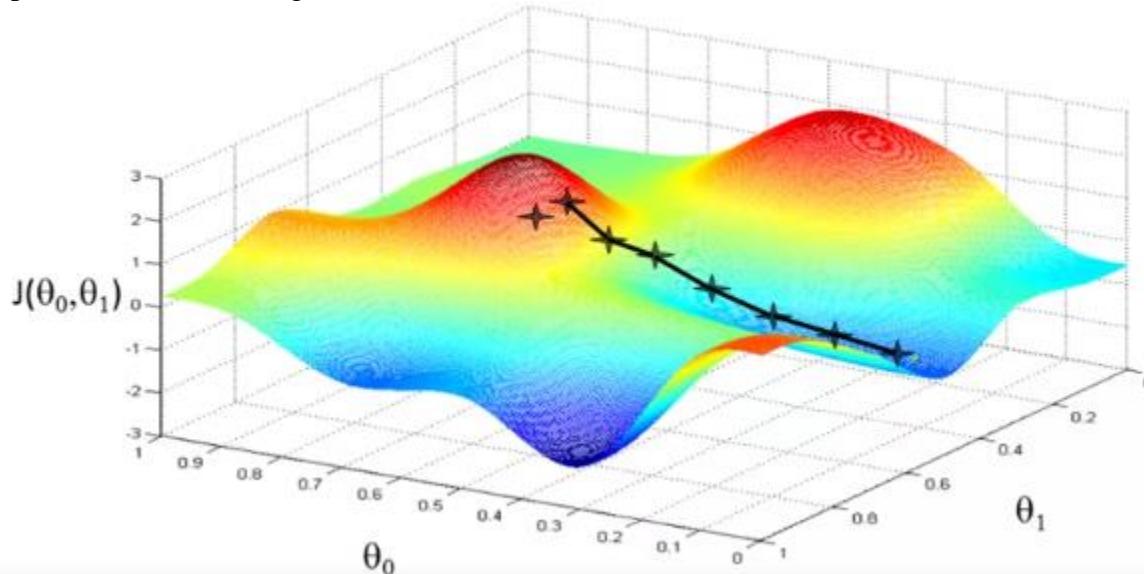


Let's say you're trying to minimize this function. So notice the axes, this is theta 0, theta 1 on the horizontal axes and J is the vertical axis and so the height of the surface shows J and we want to minimize this function. So we're going to start off with theta 0, theta 1 at some point. So imagine picking some value for theta 0, theta 1, and that corresponds to starting at some point on the surface of this function. So whatever value of theta 0, theta 1 gives you some point here. I did initialize them to 0, 0 but sometimes you initialize it to other values as well. Now, I want you to imagine that this figure shows a hole. Imagine this is like the landscape of some grassy park, with two hills like so, and I want us to imagine that you are physically standing at that point on the hill, on this little red hill in your park. In gradient descent, what we're going to do is we're going to spin 360 degrees around, just look all around us, and ask, if I were to take a little baby step in some direction, and I want to go downhill as quickly as possible, what direction do I take

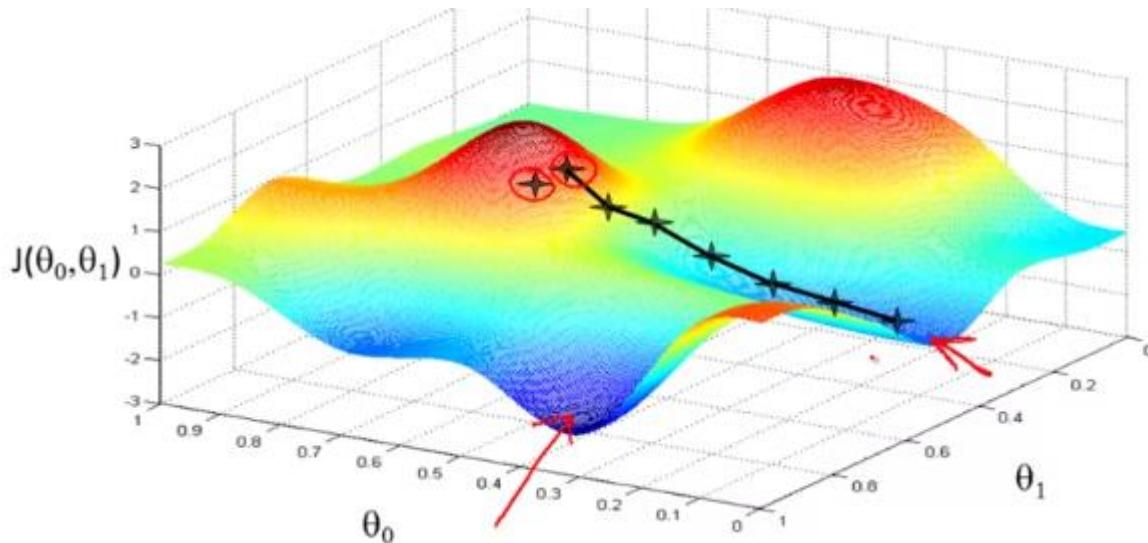
that little baby step in? If I wanna go down, so I wanna physically walk down this hill as rapidly as possible. Turns out, that if you're standing at that point on the hill, you look all around and you find that the best direction is to take a little step downhill is roughly that direction. Okay, and now you're at this new point on your hill. You're gonna, again, look all around and say what direction should I step in order to take a little baby step downhill? And if you do that and take another step, you take a step in that direction. And then you keep going. From this new point you look around, decide what direction would take you downhill most quickly. Take another step, another step, and so on until you converge to this local minimum down here.

### **Gradient descent has an interesting property.**

This first time we ran gradient descent we were starting at this point over here, right? Started at that point over here. Now imagine we had initialized gradient descent just a couple steps to the right. Imagine we'd initialized gradient descent with that point on the upper right. If you were to repeat this process, so start from that point, look all around, take a little step in the direction of steepest descent, you would do that. Then look around, take another step, and so on. And if you started just a couple of steps to the right, gradient descent would've taken you to this second local optimum over on the right.



So if you had started this first point, you would've wound up at this local optimum, but if you started just at a slightly different location, you would've wound up at a very different local optimum. And this is a property of gradient descent that we'll say a little bit more about later.



So that's the intuition in pictures. Let's look at the math.

## Gradient descent algorithm

```

repeat until convergence {
    →  $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$  (for  $j = 0$  and  $j = 1$ )
}
  }  

  learning rate
  
```

Simultaneously update  $\theta_0$  and  $\theta_1$

<b>Assignment</b> $a := b$ $\underline{a := a + 1}$	<b>Truth assertion</b> $a = b$ $a = a + 1 \times$
---	---

### Correct: Simultaneous update

```

→ temp0 :=  $\theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$ 
→ temp1 :=  $\theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$ 
→  $\theta_0 := \text{temp0}$ 
→  $\theta_1 := \text{temp1}$ 
  ↑      ↑
  
```

### Incorrect:

```

→ temp0 :=  $\theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$ 
→  $\theta_0 := \text{temp0}$ 
→ temp1 :=  $\theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$ 
→  $\theta_1 := \text{temp1}$ 
  ↑
  
```

Andrew Ng

This is the **definition of the gradient descent algorithm**: We're going to just repeatedly do this until convergence, we're going to update my parameter theta j by taking theta j and subtracting from it alpha times this term over here, okay?

So let's see, there's lot of details in this equation so let me unpack some of it.

First, this notation here,  $:=$ , gonna use  $:=$  to denote assignment, so it's the assignment operator. So briefly, if I write a  $:=$  b, what this means is, it means in a computer, this means take the value in b and use it overwrite whatever value is a. So this means set a to be equal to the value of b, which is assignment. And I can also do a  $:=$  a + 1. This means take a and increase its value by

one. Whereas in contrast, if I use the equal sign and I write  $a$  equals  $b$ , then this is a truth assertion. Okay? So if I write  $a$  equals  $b$ , then I'll asserting that the value of  $a$  equals to the value of  $b$ , right? So the left hand side, that's the computer operation, where we set the value of  $a$  to a new value. The right hand side, this is asserting, I'm just making a claim that the values of  $a$  and  $b$  are the same, and so whereas you can write  $a := a + 1$ , that means increment  $a$  by 1, hopefully I won't ever write  $a = a + 1$  because that's just wrong.  $a$  and  $a + 1$  can never be equal to the same values. Okay? So this is first part of the definition.

This **alpha here is a number that is called the learning rate**. And what alpha does is it basically controls how big a step we take downhill with creating descent. So if alpha is very large, then that corresponds to a very aggressive gradient descent procedure where we're trying take huge steps downhill and if alpha is very small, then we're taking little, little baby steps downhill. And I'll come back and say more about this later, about how to set alpha and so on.

And finally, this term here, that's a derivative term. I don't wanna talk about it right now, but I will derive this derivative term and tell you exactly what this is later, okay? And some of you will be more familiar with calculus than others, but even if you aren't familiar with calculus, don't worry about it. I'll tell you what you need to know about this term here.

Now, there's one more subtlety about gradient descent which is in gradient descent we're going to update, you know,  $\theta_0$  and  $\theta_1$ , right? So this update takes place for  $j = 0$  and  $j = 1$ , so you're gonna update  $\theta_0$  and update  $\theta_1$ . And the subtlety of how you implement gradient descent is for this expression, for this update equation, you want to simultaneously update  $\theta_0$  and  $\theta_1$ . What I mean by that is that in this equation, we're gonna update  $\theta_0 := \theta_0 - \alpha(\text{something})$  and update  $\theta_1 := \theta_1 - \alpha(\text{something})$ . And the way to implement is you should compute the right hand side, right? Compute that thing for  $\theta_0$  and  $\theta_1$  and then simultaneously, at the same time, update  $\theta_0$  and  $\theta_1$ , okay?

So let me say what I mean by that. This is a correct implementation of gradient descent meaning simultaneous update. So I'm gonna set  $\text{temp}_0$  equals that, set  $\text{temp}_1$  equals that so basic compute the right-hand sides, and then having computed the right-hand sides and stored them into variables  $\text{temp}_0$  and  $\text{temp}_1$ , I'm gonna update  $\theta_0$  and  $\theta_1$  simultaneously because that's the correct implementation.

In contrast, here's an incorrect implementation that does not do a simultaneous update. So in this incorrect implementation, we compute  $\text{temp}_0$ , and then we update  $\theta_0$ , and then we compute  $\text{temp}_1$ , and then we update  $\theta_1$ . And the difference between the right hand side and the left hand side implementations is that If you look down here, you look at this step, if by this time you've already updated  $\theta_0$ , then you would be using the new value of  $\theta_0$  to compute this derivative term. And so this gives you a different value of  $\text{temp}_1$ , than the left-hand side, right? Because you've now plugged in the new value of  $\theta_0$  into this equation. And so, this on the right-hand side is not a correct implementation of gradient descent, okay?

So I don't wanna say why you need to do the simultaneous updates. It turns out that the way gradient descent is usually implemented, which I'll say more about later, it actually turns out to be more natural to implement the simultaneous updates. And when people talk about gradient

descent, they always mean simultaneous update. If you implement the non simultaneous update, it turns out it will probably work anyway. But this algorithm wasn't right. It's not what people refer to as gradient descent, and this is some other algorithm with different properties. And for various reasons this can behave in slightly stranger ways, and so what you should do is really implement the simultaneous update of gradient descent.

So, that's the outline of the gradient descent algorithm.

In the next video, we're going to go into the details of the derivative term, which I wrote up but didn't really define. And if you've taken a calculus class before and if you're familiar with partial derivatives, it turns out that's exactly what that derivative term is, but in case you aren't familiar with calculus, don't worry about it. The next video will give you all the intuitions and will tell you everything you need to know to compute that derivative term, even if you haven't seen calculus, or even if you haven't seen partial derivatives before. And with that, with the next video, hopefully we'll be able to give you all the intuitions you need to apply gradient descent.

## Gradient Descent Intuition

In the previous video, we gave a mathematical definition of gradient descent.

Let's delve deeper and in this video get better intuition about what the algorithm is doing and why the steps of the gradient descent algorithm might make sense.

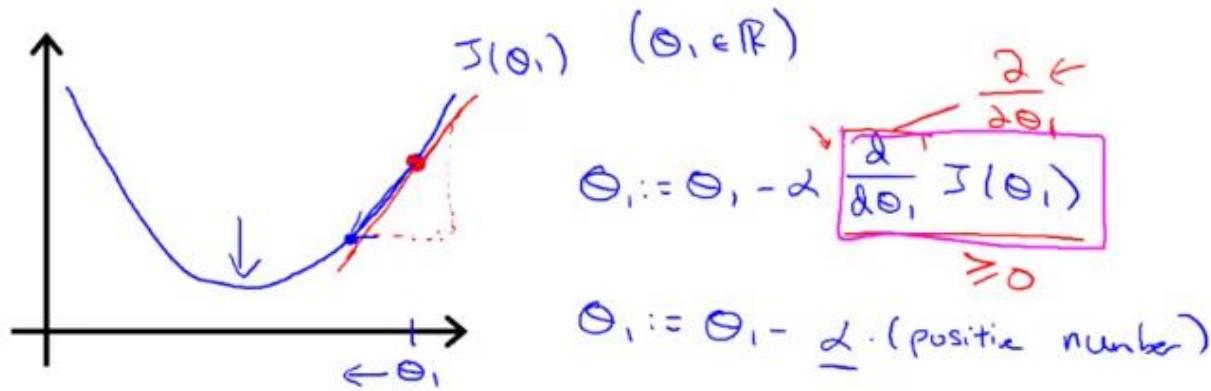
Here's a gradient descent algorithm that we saw last time and just to remind you this parameter, or this term alpha is called the learning rate. And it controls how big a step we take when updating my parameter theory j. And this second term here is the derivative term. And what I wanna do in this video is give you that intuition about what each of these two terms is doing and why when put together, this entire update makes sense.

In order to convey these intuitions, what I want to do is use a slightly simpler example, where we want to minimize the function of just one parameter. So say we have a cost function, j of just one parameter, theta one, like we did a few videos back, where theta one is a real number. So we can have one dimensional plots, which are a little bit simpler to look at.

## Gradient descent algorithm

$$\begin{aligned}
 & \text{repeat until convergence } \{ \\
 & \quad \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad (\text{simultaneously update } \\
 & \quad \} \quad \text{learning rate} \quad \text{derivative} \quad j = 0 \text{ and } j = 1) \\
 & \min_{\theta_1} J(\theta_1) \quad \theta_1 \in \mathbb{R}.
 \end{aligned}$$

Let's try to understand what gradient decent would do on this function.



So let's say, here's my function,  $J$  of  $\theta_1$ . And where  $\theta_1$  is a real number. All right? Now, let's have initialized gradient descent with  $\theta_1$  at this location.

So imagine that we start off at that point on my function. What gradient descent would do is it will update,  $\theta_1$  gets updated as  $\theta_1$  minus alpha times  $d\theta_1$   $J$  of  $\theta_1$ , right?

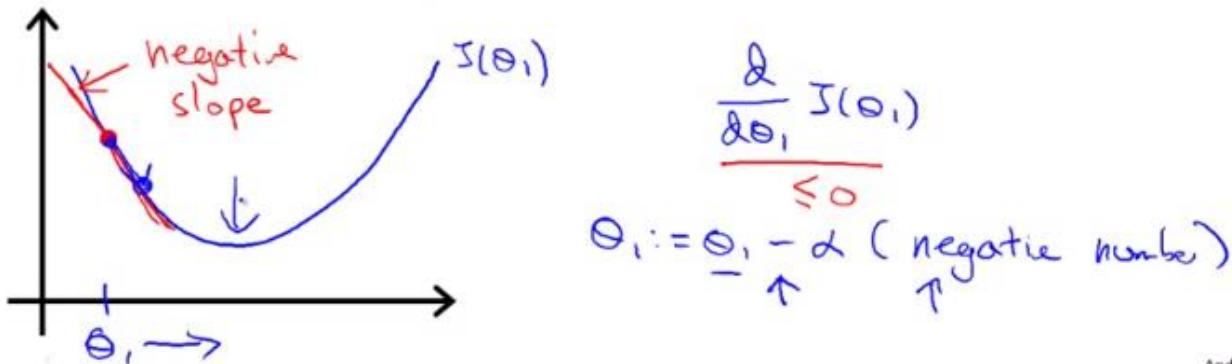
And as an aside, this derivative term, right, if you're wondering why I changed the notation from these partial derivative symbols. If you don't know what the difference is between these partial derivative symbols and the  $d\theta_1$ , don't worry about it. Technically in mathematics you call this a partial derivative and call this a derivative, depending on the number of parameters in the function  $J$ . But that's a mathematical technicality. And so for the purpose of this lecture, think of these partial symbols and  $d\theta_1$ , as exactly the same thing. And don't worry about what the real difference is. I'm gonna try to use the mathematically precise notation, but for our purposes these two notations are really the same thing.

And so let's see what this equation will do. So we're going to compute this derivative, not sure if you've seen derivatives in calculus before, but what the derivative at this point does, is basically

saying, now let's take the tangent to that point, like that straight line, that red line, is just touching this function, and let's look at the slope of this red line. That's what the derivative is, it's saying what's the slope of the line that is just tangent to the function.

Okay, the slope of a line is just this height divided by this horizontal thing. Now, this line has a positive slope, so it has a positive derivative. And so my update to theta is going to be theta 1, it gets updated as theta 1, minus alpha times some positive number. Okay. Alpha the learning rate, is always a positive number. And, so we're going to take theta one is updated as theta one minus something. So I'm gonna end up moving theta one to the left. I'm gonna decrease theta one, and we can see this is the right thing to do cuz I actually wanna head in this direction. You know, to get me closer to the minimum over there. So, gradient descent so far says we're going the right thing.

Let's look at another example.



So let's take my same function  $J$ , let's try to draw from the same function,  $J$  of theta 1. And now, let's say I had to say initialize my parameter over there on the left. So theta 1 is here. I glare at that point on the surface. Now my derivative term dd theta one  $J$  of theta one when you evaluate at this point, we're gonna look at right the slope of that line, so this derivative term is a slope of this line. But this line is slanting down, so this line has negative slope. Right. Or alternatively, I say that this function has negative derivative, just means negative slope at that point. So this is less than equals to 0, so when I update theta, I'm gonna have theta. Just update this theta of minus alpha times a negative number. And so I have theta 1 minus a negative number which means I'm actually going to increase theta, because it's minus of a negative number, means I'm adding something to theta. And what that means is that I'm going to end up increasing theta until it's not here, and increase theta wish again seems like the thing I wanted to do to try to get me closer to the minimum.

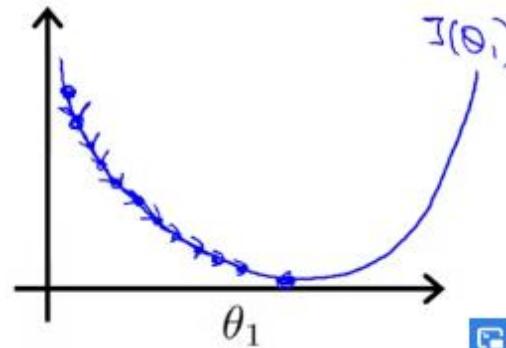
So this hopefully explains the intuition behind what a derivative is doing, let's take a look at the learning rate term alpha and see what that's doing. So here's my gradient descent update rule, that's this equation.

And let's look at what could happen if alpha is either too small or if alpha is too large.

So this first example, what happens if alpha is too small?

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

If  $\alpha$  is too small, gradient descent can be slow.

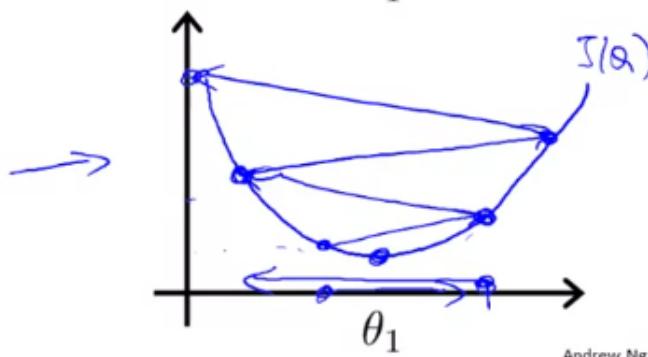


So here's my function  $J$ ,  $J$  of theta. Let's all start here. If alpha is too small, then what I'm gonna do is gonna multiply my update by some small number, so end up taking a baby step like that. Okay, so this one step. Then from this new point, I'm gonna have to take another step. But if alpha's too small, I take another little baby step. And so if my learning rate is too small I'm gonna end up taking these tiny tiny baby steps as you try to get to the minimum. And I'm gonna need a lot of steps to get to the minimum and so if alpha is too small gradient descent can be slow because it's gonna take these tiny tiny baby steps and so it's gonna need a lot of steps before it gets anywhere close to the global minimum.

### Now how about if our alpha is too large?

So, here's my function  $J$  of theta. It turns out that if alpha's too large, then gradient descent can overshoot the minimum and may even fail to converge or even diverge, so here's what I mean.

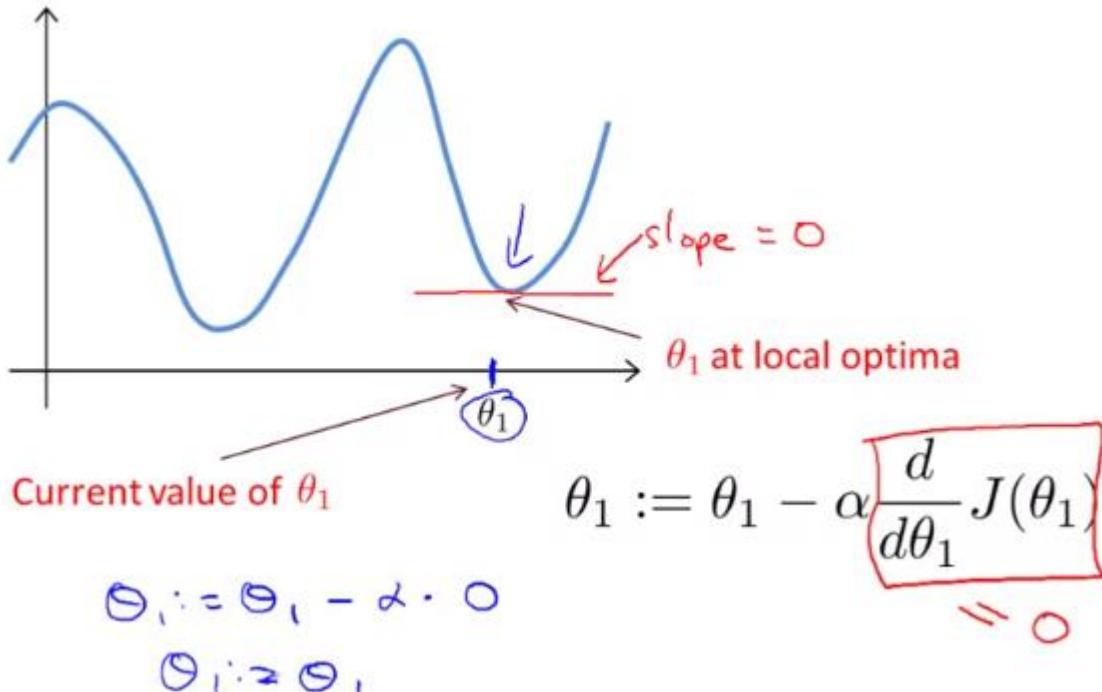
If  $\alpha$  is too large, gradient descent can overshoot the minimum. It may fail to converge, or even diverge.



Let's say it's all our data there, it's actually close to minimum. So the derivative points to the right, but if alpha is too big, I want to take a huge step. Remember, take a huge step like that. So it ends up taking a huge step, and now my cost functions is not going to work. Cuz it starts off with this value, and now, my values have gone in reverse. Now my derivative points go to the left, it says I should decrease data. But if my learning is too big, I may take a huge step going from here all the way to out there. So we end up being over there, right? And if my learning rate is too big, we can take another huge step on the next elevation and kind of overshoot and overshoot and so on, until you already notice I'm actually getting further and further away from the minimum. So if alpha is too large, it can fail to converge or even diverge.

Now, I have another question for you.

So this is 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 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 solution at the local optimum.

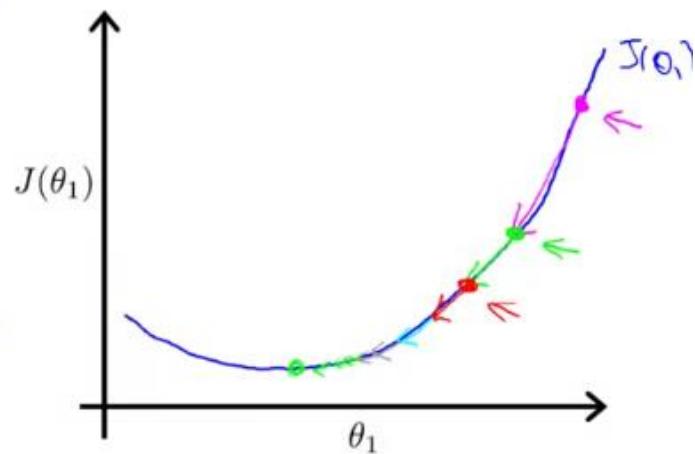
This also explains why gradient descent can converge to the local minimum even with the learning rate alpha fixed. Here's what I mean by that let's look in the example.

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.

**Gradient descent can converge to a local minimum, even with the learning rate  $\alpha$  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  $\alpha$  over time.



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 to zero, as I approach the minimum. So after one step of descent, my new derivative is a little bit smaller. So I wanna take another step in the gradient descent. I will naturally take a somewhat smaller step from this green point right there from the magenta point. Now with a new point, a red point, and I'm even closer to global minimum so the derivative here will be even smaller than it was at the green point. So I'm gonna another step in the gradient descent. Now, my derivative term is even smaller and so the magnitude of the update to theta one is even smaller, so take a small step like so. And as gradient descent runs, you will automatically take smaller and smaller steps. Until eventually you're taking very small steps, you know, and you finally converge to the local minimum.

So just to recap, in gradient descent as we approach a local minimum, gradient descent will automatically take smaller steps. And that's because as we approach the local minimum, by definition the local minimum is when the derivative is equal to zero. As we approach local minimum, this derivative term will automatically get smaller, and so gradient descent will automatically take smaller steps. **This is why actually no need to decrease alpha over time.**

So that's the **gradient descent algorithm** and you can use it to try to minimize any cost function **J**, not the cost function **J** that we defined for linear regression. In the next video, we're going to take the function **J** and set that back to be exactly linear regression's cost function, the square cost function that we came up with earlier. And taking gradient descent and this square cost function and putting them together. That will give us our first learning algorithm, that'll give us a linear regression algorithm.

## Gradient Descent for Linear Regression

In previous videos, we talked about the gradient descent algorithm and we talked about the linear regression model and the squared error cost function.

In this video we're gonna put together gradient descent with our cost function, and that will give us an algorithm for linear regression or putting a straight line to our data.

So this was what we worked out in the previous videos.

This gradient descent algorithm which you should be familiar and here's the linear regression model with our linear hypothesis and our squared error cost function. **What we're going to do is apply gradient descent to minimize our squared error cost function.**

### Gradient descent algorithm

```
repeat until convergence {
     $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$ 
    (for  $j = 1$  and  $j = 0$ )
}
```

### Linear Regression Model

$$h_{\theta}(x) = \theta_0 + \theta_1 x$$

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

Now in order to apply gradient descent, in order to, you know, write this piece of code, the key term we need is this derivative term over here.

### Gradient descent algorithm

```
repeat until convergence {
     $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$ 
    (for  $j = 1$  and  $j = 0$ )
}
```

### Linear Regression Model

$$h_{\theta}(x) = \theta_0 + \theta_1 x$$

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

$$\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$$

So you need to figure out what is this partial derivative term and plugging in the definition of the cost function  $J$ , this turns out to be this.

$$\begin{aligned} \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) &= \frac{2}{2m} \cdot \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 \\ &= \frac{2}{2m} \cdot \frac{1}{2m} \sum_{i=1}^m (\underline{\theta_0 + \theta_1 x^{(i)}} - y^{(i)})^2 \end{aligned}$$

Sum from  $y$  equals 1 though  $m$  of this squared error cost function term. And all I did here was I just, you know plug in the definition of the cost function there. And simplifying a little bit more, this turns out to be equal to this. Sigma  $i$  equals one through  $m$  of theta zero plus theta one  $x(i)$  minus  $y(i)$  squared. And all I did there was I took the definition for my hypothesis and plugged it in there.

And turns out we need to figure out what is this partial derivative for two cases, for  $J$  equals 0 and for  $J$  equals 1. So we want to figure out what is this partial derivative for both the theta 0 case and the theta 1 case. And I'm just going to write out the answers.

$$\begin{aligned} \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) &= \frac{2}{2m} \cdot \frac{1}{2m} \sum_{i=1}^m (\underline{h_\theta(x^{(i)})} - \underline{y^{(i)}})^2 \\ &= \frac{2}{2m} \cdot \frac{1}{2m} \sum_{i=1}^m (\underline{\theta_0 + \theta_1 x^{(i)}} - \underline{y^{(i)}})^2 \\ \Theta_0, j = 0 : \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) &= \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \\ \Theta_1, j = 1 : \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) &= \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)} \end{aligned}$$

It turns out this first term is, simplifies to  $1/m$  sum from over my training set of just that of  $X(i)$ - $Y(i)$  and for this term partial derivative let's write the theta 1, it turns out I get this term. Minus  $Y(i)$  times  $X(i)$ .

Okay and computing these partial derivatives, so we're going from this equation. Right going from this equation to either of the equations down there. Computing those partial derivative terms requires some multivariate calculus. If you know calculus, feel free to work through the derivations yourself and check that if you take the derivatives, you actually get the answers that I got.

But if you're less familiar with calculus, don't worry about it and it's fine to just take these equations that were worked out and you won't need to know calculus or anything like that, in order to do the homework so let's implement gradient descent and get back to work. So armed with these definitions or armed with what we worked out to be the **derivatives which is really just the slope of the cost function** we can now plug them back in to our gradient descent algorithm.

## Gradient descent algorithm

repeat until convergence {

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

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

}

$$\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$

update  
 $\theta_0$  and  $\theta_1$   
simultaneously

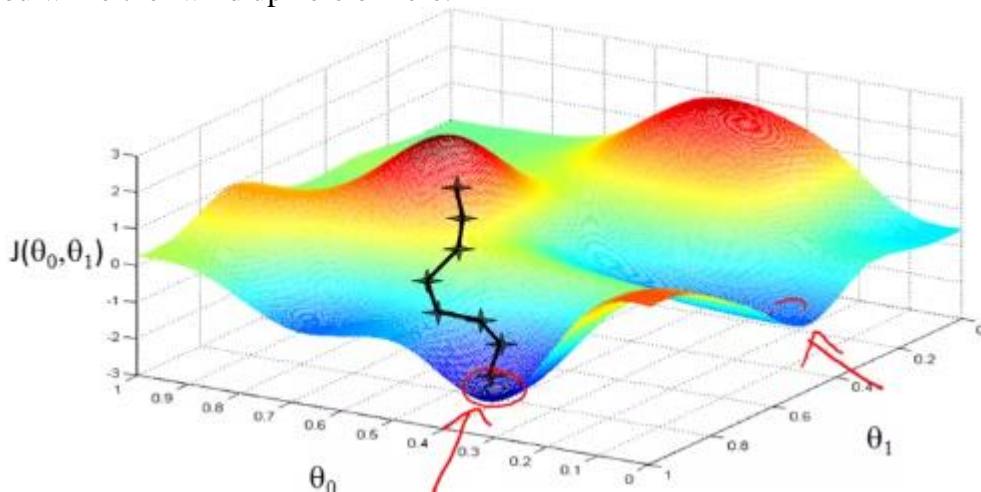
$$\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$

So here's gradient descent for linear regression which is gonna repeat until convergence, **theta 0 and theta 1 get updated** as you know this thing minus alpha times the derivative term. So this term here. So here's our linear regression algorithm. This first term here. That term is of course just the partial derivative with respect to theta zero, that we worked out on a previous slide. And this second term here, that term is just a partial derivative in respect to theta 1, that we worked out on the previous line.

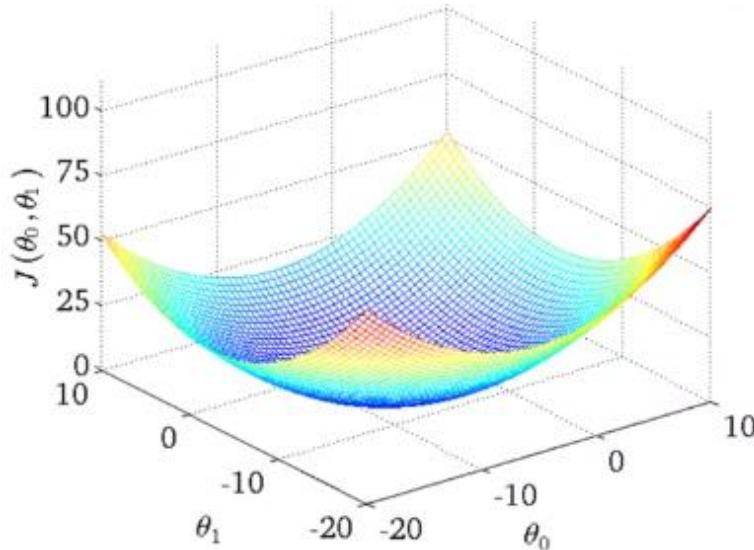
And just as a quick reminder, you must, when implementing gradient descent, there's actually this detail that you should be implementing it so the **update theta 0 and theta 1 simultaneously**.

So. Let's see how gradient descent works.

One of the issues we saw with gradient descent is that it can be susceptible to local optima. So when I first explained gradient descent I showed you this picture of it going downhill on the surface, and we saw how depending on where you initialize it, you can end up at different local optima. You will either wind up here or here.



But, it turns out that the **cost function for linear regression is always going to be a bowl shaped function** like this. The technical term for this is that this is called a convex function.

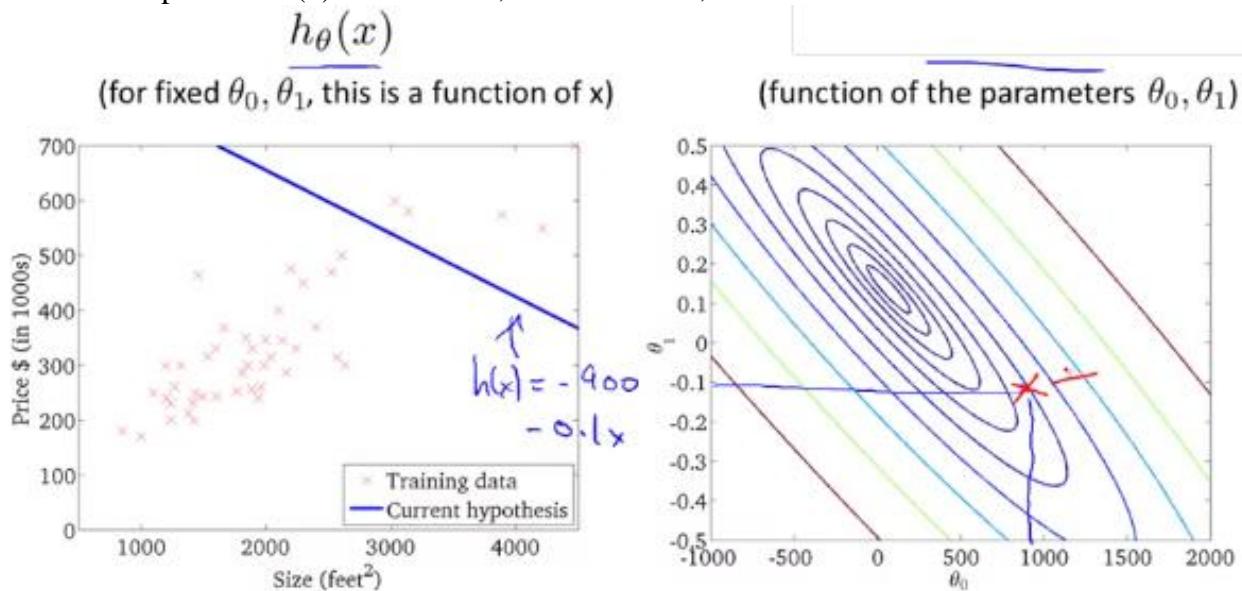


And I'm not gonna give the formal definition for what is a convex function, C, O, N, V, E, X. But informally a convex function means a bowl shaped function and so this function doesn't have any local optima except for the one global optimum.

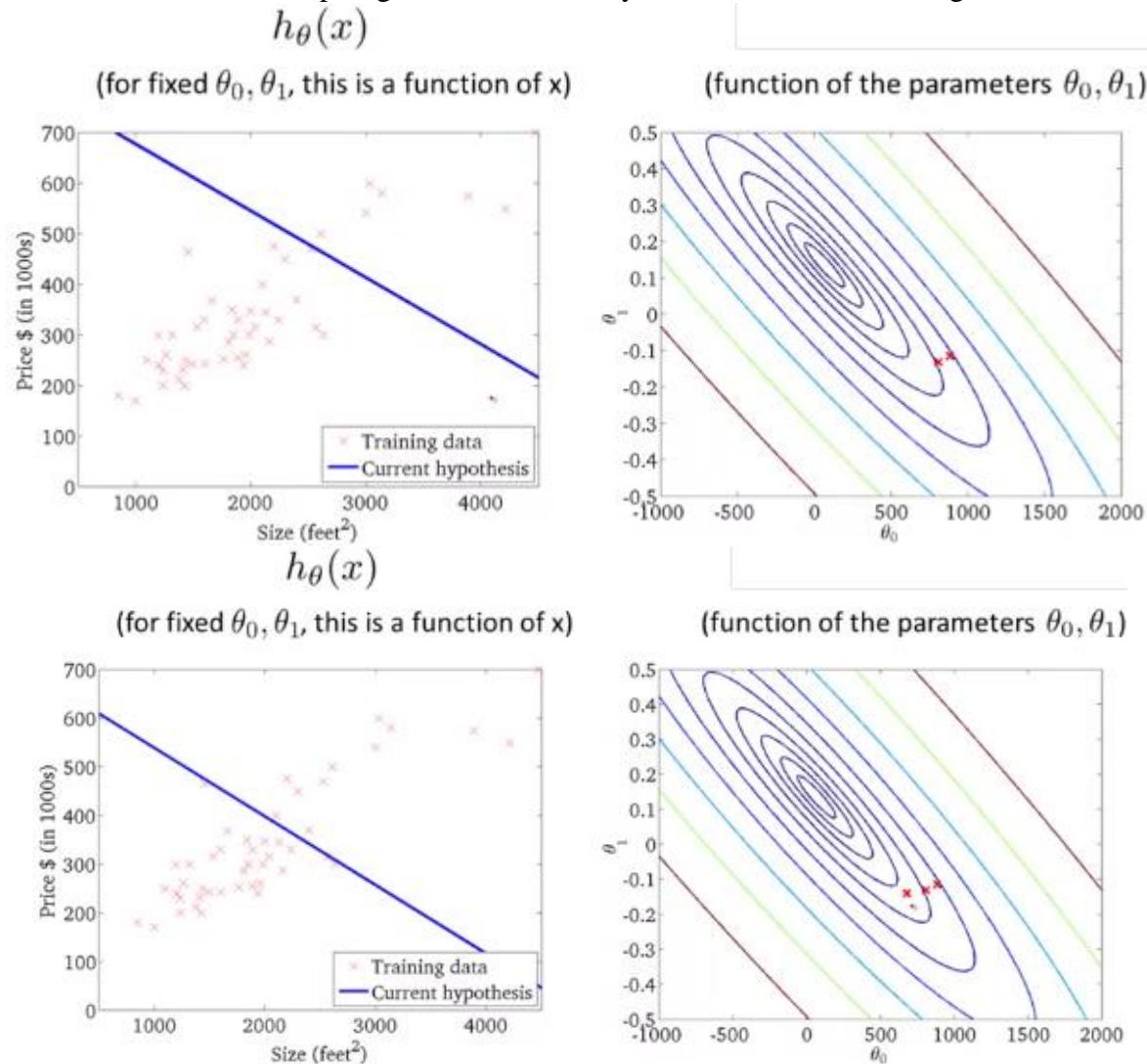
And does gradient descent on this type of cost function which you get whenever you're using linear regression it will always converge to the global optimum. Because there are no other local optimum, global optimum.

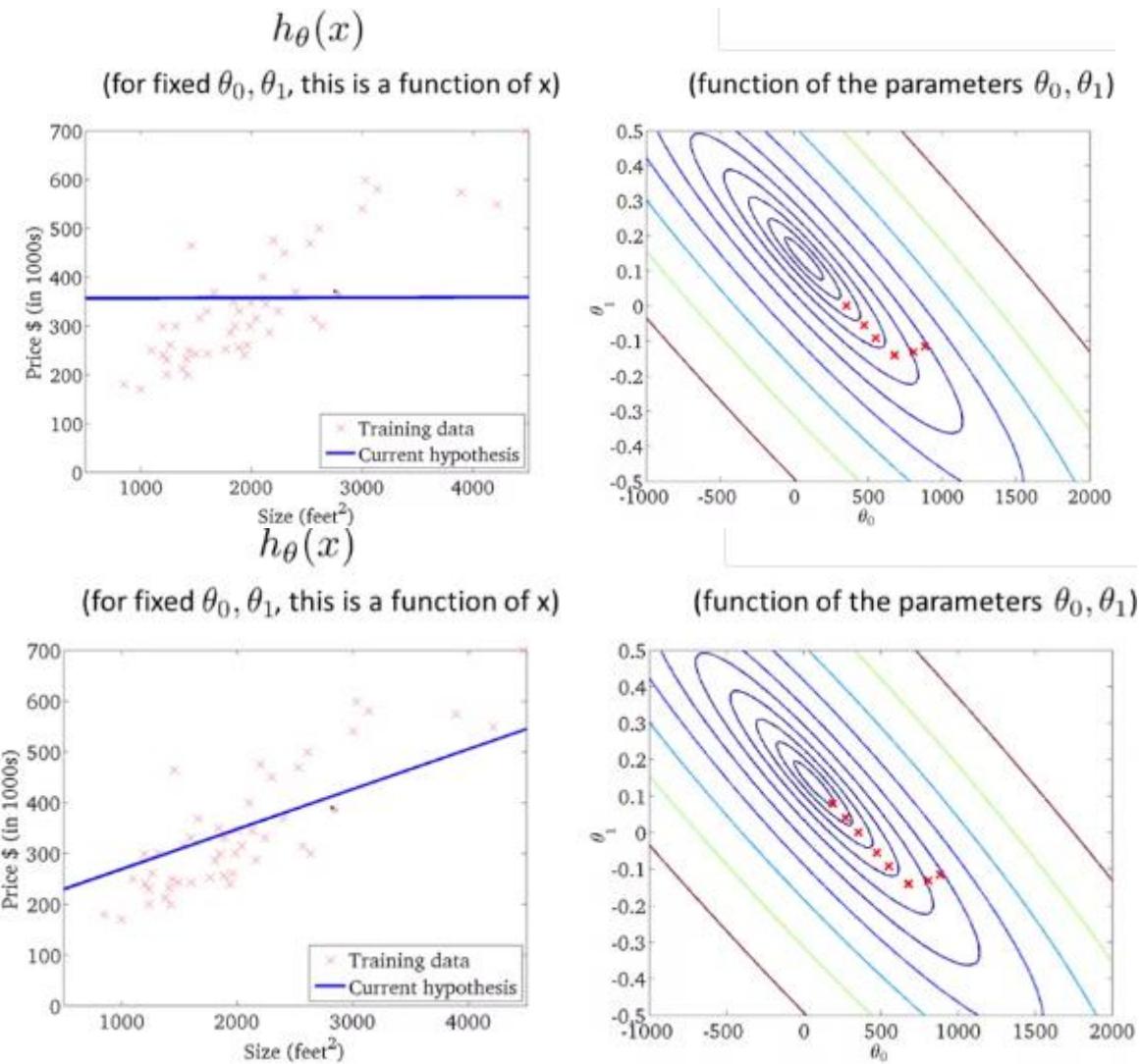
So now let's see this **algorithm in action**.

As usual, here are plots of the hypothesis function and of my cost function  $j$ . And so let's say I've initialized my parameters at this value. Let's say, usually you initialize your parameters at zero, zero. Theta zero and theta one equals zero. But for illustration, in this physical interpretation I've initialized you know, theta zero at 900 and theta one at about -0.1 okay. And so this corresponds to  $h(x) = 900 - 0.1x$ , this is this line, out here on the cost function.



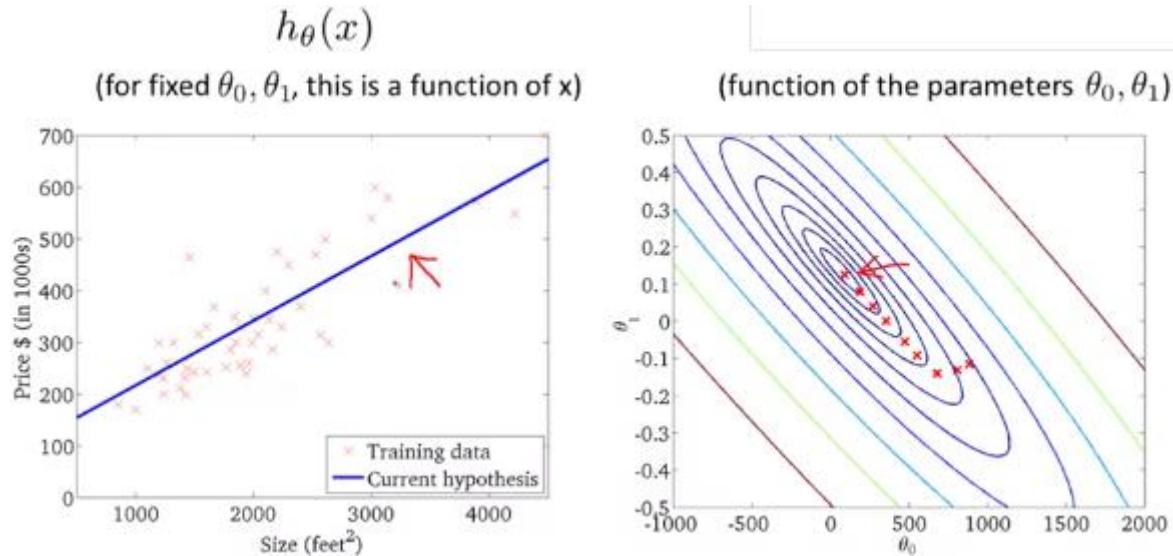
Now, if we take one step in gradient descent, we end up going from this point out here, over to the down and left, to that second point over there. And you notice that my line changed a little bit, and as I take another step of gradient descent, my line on the left will change.





Right?

And I've also moved to a new point on my cost function. And as I take further steps of gradient descent, I'm going down in cost. So my parameters and such are following this trajectory. And if you look on the left, this corresponds with hypotheses. That seem to be getting to be better and better fits to the data until **eventually I've now wound up at the global minimum** and this global minimum corresponds to this hypothesis, which gets me a good fit to the data.



And so that's gradient descent, and we've just run it and gotten a good fit to my data set of housing prices. And you can now use it to predict, you know, if your friend has a house size 1250 square feet, you can now read off the value and tell them that I don't know maybe they could get \$250,000 for their house.

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.

## "Batch" Gradient Descent

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

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

So ever step of gradient descent we end up computing something like this that sums over our m training examples and so the term batch gradient descent refers to the fact that we're looking at the entire batch of training examples. And again, it's really not a great name, but this is what machine learning people call it. And it turns out that there are sometimes other versions of gradient descent that are not batch versions, but they instead do not look at the entire training set but look at small subsets of the training sets at a time. And we'll talk about those versions later in this course as well. But for now using the algorithm we just learned about or **using batch gradient descent you now know how to implement gradient descent for linear regression.**

So that's linear regression with gradient descent.

If you've seen advanced linear algebra before, so some of you may have taken a class in **advanced linear algebra**. You might know that **there exists a solution** for numerically solving for the minimum of the cost function  $j$  without needing to use an iterative algorithm like gradient descent.

Later in this course we'll talk about 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 learning algorithm.

We'll later have exercises in which we'll ask you to implement gradient descent and hopefully see these algorithms right for yourselves.

But before that I first want to tell you in the next set of videos.

The first one to tell you **about a generalization of the gradient descent algorithm that will make it much more powerful**. And I guess I'll tell you about that in the next video.

## Linear Algebra Review

### Matrices and Vectors

Let's get started with our linear algebra review.

In this video I want to tell you **what are matrices** and what are vectors.

A matrix is a rectangular array of numbers written between square brackets. So, for example, here is a matrix on the right, a left square bracket. And then, write in a bunch of numbers.

These could be features from a learning problem or it could be data from somewhere else, but the specific values don't matter, and then I'm going to close it with another right bracket on the right. And so that's one matrix. And, here's another example of the matrix, let's write 1 2 3 4 5 6. So matrix is just another way for saying, is a 2D or a two dimensional array.

## Matrix: Rectangular array of numbers:

$$\begin{array}{c} \rightarrow \\ \rightarrow \\ \rightarrow \\ \rightarrow \end{array} \left[ \begin{array}{cc} 1402 & 191 \\ 1371 & 821 \\ 949 & 1437 \\ 147 & 1448 \end{array} \right] \quad \begin{array}{c} \rightarrow \\ \rightarrow \end{array} \left[ \begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \end{array} \right]$$

$\uparrow \quad \uparrow$

$4 \times 2 \text{ matrix}$

$\rightarrow [R^{4 \times 2}]$

$\uparrow \quad \uparrow \quad \uparrow$

$2 \times 3 \text{ matrix}$

$\rightarrow [R^{2 \times 3}]$

### Dimension of matrix: number of rows x number of columns

And the other piece of knowledge that we need is that the dimension of the matrix is going to be written as the number of row times the number of columns in the matrix.

So, concretely, this example on the left, this has 1, 2, 3, 4 rows and has 2 columns, and so this example on the left is a 4 by 2 matrix - number of rows by number of columns. So, four rows, two columns.

This one on the right, this matrix has two rows. That's the first row, that's the second row, and it has three columns. That's the first column, that's the second column, that's the third column. So, this second matrix we say it is a 2 by 3 matrix. So we say that the dimension of this matrix is 2 by 3.

Sometimes you also see this written out, in the case of left, you will see this written out as  $R^{4 \times 2}$  or concretely what people will sometimes say this matrix is an element of the set  $R^{4 \times 2}$ .

So, this thing here, this just means the set of all matrices that of dimension  $4 \times 2$  and this thing on the right, sometimes this is written out as a matrix that is an  $R^{2 \times 3}$ . So if you ever see,  $2 \times 3$ . So if you ever see something like this are  $4 \times 2$  or are  $2 \times 3$ , people are just referring to matrices of a specific dimension.

Next, let's talk about how to refer to specific elements of the matrix. And by matrix elements, other than the matrix I just mean the entries, so the numbers inside the matrix. So, in the standard notation, if  $A$  is this matrix here, then  $A_{ij}$  is going to refer to the  $i, j$  entry, meaning the entry in the matrix in the  $i$ th row and the  $j$ th column. So for example  $A_{11}$  is going to refer to the entry in the 1st row and the 1st column, so that's the first row and the first column and so  $A_{11}$  is going to be equal to 1402.

## Matrix Elements (entries of matrix)

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

$A_{ij}$  = "i, j entry" in the i<sup>th</sup> row, j<sup>th</sup> column.

$$A_{11} = 1402$$

$$A_{12} = 191$$

$$A_{32} = 1437$$

$$A_{41} = 147$$

~~$A_{43}$~~  = undefined (error)

Another example,  $A_{12}$  is going to refer to the entry in the first row and the second column and so  $A_{12}$  is going to be equal to one nine one 191. This come from a quick examples. Let's see, A, oh let's say  $A_{32}$ , is going to refer to the entry in the 3rd row, and second column, right, because that's 3 2 so that's equal to 1437. And finally,  $A_{41}$  is going to refer to this one right, fourth row, first column is equal to 147. And if, hopefully you won't, but if you were to write and say well this  $A_{43}$ , well, that refers to the fourth row, and the third column that, you know, this matrix has no third column so this is undefined, you know, or you can think of this as an error. There's no such element as  $A_{43}$ , so, you know, you shouldn't be referring to  $A_{43}$ .

So, the matrix gets you a way of letting you quickly organize, index and access lots of data. In case I seem to be tossing up a lot of concepts, a lot of new notations very rapidly, you don't need to memorize all of this, but on the course website where we have posted the lecture notes, we also have all of these definitions written down. So you can always refer back, you know, either to these slides, possible coursework, so audible lecture notes if you forget well,  $A_{41}$  was what? Which row, which column was that? Don't worry about memorizing everything now. You can always refer back to the written materials on the course website, and use that as a reference. So that's what a matrix is.

Next, let's talk about **what is a vector**.

A vector turns out to be a **special case of a matrix**. A vector is a matrix that **has only 1 column** so you have an  $N \times 1$  matrix, then that's a, remember, right? N is the number of rows, and 1 here is the number of columns, so, so **matrix with just one column is what we call a vector**.

So here's an example of a vector, with I guess I have N equals four elements here. So we also call this thing, another term for this is a **four dimensional vector**, just means that this is a **vector with four elements**, with four numbers in it. And, just as earlier for matrices you saw this

notation  $R^{3 \times 2}$  to refer to 3 by 2 matrices, for this vector we are going to refer to this as a vector in the set  $R^4$ . So this  $R^4$  means a set of four-dimensional vectors.

Next let's talk about how to refer to the elements of the vector. We are going to use the notation  $y_i$  to refer to the  $i$ th element of the vector  $y$ . So if  $y$  is this vector,  $y$  subscript  $i$  is the  $i$ th element. So  $y_1$  is the first element, four sixty,  $y_2$  is equal to the second element, two thirty two, there's the first, there's the second.  $y_3$  is equal to 315 and so on, and only  $y_1$  through  $y_4$  are defined because this is a 4-dimensional vector.

Also it turns out that there are actually 2 conventions for how to index into a vector and here they are. **Sometimes, people will use one index and sometimes zero index factors.**

So this example on the left is a one in that specter where the element we write is  $y_1, y_2, y_3, y_4$ . And this example in the right is an example of a zero index vector where we start the indexing of the elements from zero. So the elements go from  $y$  zero up to  $y$  three. And this is a little bit like the arrays of some primary languages where the arrays can either be indexed starting from one, the first element of an array is sometimes a  $y_1$ , this is sequence notation I guess, and sometimes it's zero index depending on what programming language you use. So it turns out that **in most of math, the one index version is more common.** For a lot of machine learning applications, zero index vectors gives us a more convenient notation.

So what you should usually do is, unless otherwise specified, you should assume we are using one index vectors. In fact, **throughout the rest of these videos on linear algebra review, I will be using one index vectors.** But just be aware that when we are talking about machine learning applications, sometimes I will explicitly say when we need to switch to, when we need to use the **zero index vectors** as well.

Finally, by convention, usually when writing matrices and vectors, most people will **use upper case to refer to matrices.** So we're going to use capital letters like A, B, C, you know, X, to refer to matrices, and usually **we'll use lowercase, like a, b, x, y, to refer to either numbers, or just real numbers or scalars or to vectors.**

This isn't always true but this is the more common notation where we use lower case "Y" for referring to vector and we usually use upper case to refer to a matrix. So, you now know what are matrices and vectors.

Next, we'll talk about some of the things you can do with them

## Addition and Scalar Multiplication

In this video we'll talk about matrix **addition and subtraction**, as well as how to **multiply a matrix by a number**, also called **Scalar Multiplication**.

Let's start an example. Given two matrices like these, let's say I want to add them together. How do I do that? And so, what does addition of matrices mean? It turns out that if you want to add two matrices, what you do is you just add up the elements of these matrices one at a time. So, my result of adding two matrices is going to be itself another matrix and the first element again just

by taking one and four and adding them together, so I get five. The second element I get by taking two and two and adding them, so I get four; three plus three plus zero is three, and so on. I'm going to stop changing colors, I guess. And, on the right is five, ten and two. And it turns out **you can add only two matrices that are of the same dimensions.**

## Matrix Addition

$$\begin{array}{c}
 \begin{array}{ccc}
 & \downarrow & \downarrow \\
 \rightarrow & \boxed{1} & 0 \\
 \rightarrow & \boxed{2} & 5 \\
 \rightarrow & \boxed{3} & 1
 \end{array} & + & 
 \begin{array}{cc}
 \boxed{4} & 0.5 \\
 \boxed{2} & 5 \\
 \boxed{0} & 1
 \end{array} & = & 
 \begin{array}{cc}
 \boxed{5} & 0.5 \\
 \boxed{4} & 10 \\
 \boxed{3} & 2
 \end{array} \\
 \hline
 \begin{matrix} 3 \times 2 \\ \text{matrix} \end{matrix} & & \begin{matrix} 3 \times 2 \\ \end{matrix} & & \begin{matrix} 3 \times 2 \\ \end{matrix}
 \end{array}$$
  

$$\begin{array}{c}
 \begin{array}{cc}
 \cancel{\boxed{1}} & 0 \\
 \cancel{\boxed{2}} & 5 \\
 \cancel{\boxed{3}} & 1
 \end{array} & + & 
 \begin{array}{cc}
 \cancel{\boxed{4}} & 0.5 \\
 \cancel{\boxed{2}} & 5 \\
 \cancel{\boxed{0}} & 1
 \end{array} & = & \text{error}
 \end{array}$$

So this example is a three by two matrix, because this has 3 rows and 2 columns, so it's 3 by 2. This is also a 3 by 2 matrix, and the result of adding these two matrices is a 3 by 2 matrix again. So you can only add matrices of the same dimension, and the result will be another matrix that's of the same dimension as the ones you just added.

Where as in contrast, if you were to take these two matrices, so this one is a 3 by 2 matrix, okay, 3 rows, 2 columns. This here is a 2 by 2 matrix. And **because these two matrices are not of the same dimension, you know, this is an error, so you cannot add these two matrices and, you know, their sum is not well-defined.** So that's matrix addition.

**Next, let's talk about multiplying matrices by a scalar number.** And the scalar is just a, maybe an overly fancy term for, you know, a number or a real number. Alright, this means real number.

## Scalar Multiplication

real number

$$\begin{array}{c}
 3 \times \left[ \begin{array}{cc} 1 & 0 \\ 2 & 5 \\ 3 & 1 \end{array} \right] = \left[ \begin{array}{cc} 3 & 0 \\ 6 & 15 \\ 9 & 3 \end{array} \right] = \left[ \begin{array}{cc} 1 & 0 \\ 2 & 5 \\ 3 & 1 \end{array} \right] \times 3 \\
 \boxed{3 \times 2} \qquad \qquad \qquad \boxed{3 \times 2} \qquad \qquad \qquad \boxed{3 \times 2}
 \end{array}$$
  

$$\left[ \begin{array}{cc} 4 & 0 \\ 6 & 3 \end{array} \right] / 4 = \frac{1}{4} \left[ \begin{array}{cc} 4 & 0 \\ 6 & 3 \end{array} \right] = \left[ \begin{array}{cc} 1 & 0 \\ \frac{3}{2} & \frac{3}{4} \end{array} \right]$$

So let's take the number 3 and multiply it by this matrix. And if you do that, the result is pretty much what you'll expect. You just take your elements of the matrix and multiply them by 3, one at a time. So, you know, one times three is three. What, two times three is six, 3 times 3 is 9, and let's see, I'm going to stop changing colors again. Zero times 3 is zero. Three times 5 is 15, and 3 times 1 is three. And so this matrix is the result of multiplying that matrix on the left by 3. And you notice, again, this is a 3 by 2 matrix and the result is a matrix of the same dimension.

This is a 3 by 2, both of these are 3 by 2 dimensional matrices. And by the way, you can write multiplication, you know, either way. So, I have three times this matrix. I could also have written this matrix as 1, 0, 2, 5, 3, 1, right. I just copied this matrix over to the right. I can also take this matrix and multiply this by three. So whether it's you know, 3 times the matrix or the matrix times three is the same thing and this thing here in the middle is the result.

You can also take a matrix and **divide it by a number**. So, turns out taking this matrix and dividing it by four, this is actually the same as taking the number one quarter, and multiplying it by this matrix. 4, 0, 6, 3 and so, you can figure the answer, the result of this product is, one quarter times four is one, one quarter times zero is zero. One quarter times six is, what, three halves, about six over four is three halves, and one quarter times three is three quarters. And so that's the result of computing this matrix divided by four. That give you the result.

Finally, for a **slightly more complicated example**, you can also take these operations and combine them together.

## Combination of Operands

Scalar multiplication

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

Scalar division

$= \begin{bmatrix} 3 \\ 12 \\ 6 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 5 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \\ \frac{2}{3} \end{bmatrix}$

matrix subtraction / vector subtraction

matrix addition / vector addition

$= \begin{bmatrix} 2 \\ 12 \\ 10\frac{1}{3} \end{bmatrix}$

3x1 matrix  
3-dimensional vector

Andrew Ng

So in this calculation, I have three times a vector plus a vector minus another vector divided by three. So just make sure we know where these are, right. This multiplication. This is an example of scalar multiplication because I am taking three and multiplying it. And this is, you know, another scalar multiplication. Or more like scalar division, I guess. It really just means one zero times this. And so if we evaluate these two operations first, then what we get is this thing is equal to, let's see, so three times that vector is three, twelve, six, plus my vector in the middle which is a 0 0 5 minus one, zero, two-thirds, right?

And again, just to make sure we understand what is going on here, this plus symbol, that is matrix addition, right? I really, since these are vectors, remember, vectors are special cases of matrices, right? This, you can also call this vector addition. This minus sign here, this is again a matrix subtraction, but because this is an n by 1, really a three by one matrix, that this is actually a vector, so this is also vector, this column. We call this matrix a vector subtraction, as well. OK?

And finally to wrap this up. This therefore gives me a vector, whose first element is going to be 3+0-1, so that's 3-1, which is 2. The second element is 12+0-0, which is 12. And the third element of this is, what, 6+5-(2/3), which is 11-(2/3), so that's 10 and one-third and see, you close this square bracket. And so this gives me a 3 by 1 matrix, which is also just called a 3 dimensional vector, which is the outcome of this calculation over here.

So that's how you add and subtract matrices and vectors and multiply them by scalars or by row numbers.

So far I have only talked about how to multiply matrices and vectors by scalars, by row numbers. In the next video we will talk about a much more interesting step, of taking 2 matrices and multiplying 2 matrices together.

## Matrix Vector Multiplication

In this video, I'd like to start talking about how to multiply together two matrices. We'll start with a special case of that, of **matrix vector multiplication** - multiplying a matrix together with a vector.

Let's start with an example. Here is a matrix, and here is a vector, and let's say we want to multiply together this matrix with this vector, what's the result?

Let me just work through this example and then we can step back and look at just what the steps were. It turns out the **result of this multiplication process is going to be, itself, a vector**. And I'm just going work with this first and later we'll come back and see just what I did here.

### Example

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

$3 \times 2 \quad 2 \times 1 \quad 3 \times 1 \text{ matrix}$

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

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

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

To get the first element of this vector I am going to take these two numbers and multiply them with the first row of the matrix and add up the corresponding numbers. Take 1 multiplied by 1, and take 3 and multiply it by 5, and that's what, that's one plus fifteen so that gives me sixteen. I'm going to write sixteen here. Then for the second row, second element, I am going to take the second row and multiply it by this vector, so I have 4 times 1, plus 0 times 5, which is equal to four, so you'll have four there. And finally for the last one I have two times one, so 2 by 1, plus one by 5, which is equal to a 7, and so I get a 7 over there. It turns out that the results of multiplying that's a **3x2 matrix by a 2x1 matrix, which is also just a two-dimensional vector**, the result of this is going to be a **3x1 matrix**, so that's why three by one **3x1 matrix**, in other words a **3x1 matrix is just a three dimensional vector**.

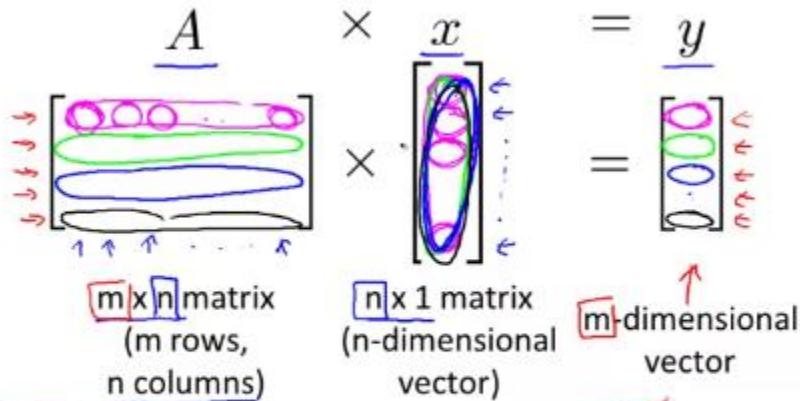
So I realize that I did that pretty quickly, and you're probably not sure that you can repeat this process yourself, but let's look in more detail at what just happened and what this process of multiplying a matrix by a vector looks like.

Here's the details of how to multiply a matrix by a vector. Let's say I have a matrix A and want to multiply it by a vector x. The result is going to be some vector y. So the matrix A here is a **m by n dimensional matrix**, so m rows and n columns and we are going to multiply that by a **n by 1 matrix**, in other words an **n dimensional vector**. It turns out this "n" here has to match this "n" here. In other words, the number of columns in this matrix, so it's the number of n columns, **the**

**number of columns here has to match the number of rows here.** It has to match the dimension of this vector. And the **result of this product is going to be an n-dimensional vector y.** In other words, the number of rows here in the resultant vector, "m" is going to be equal to the number of rows in this matrix "A".

So how do you actually compute this vector "y"?

### Details:



To get  $y_i$ , multiply  $A$ 's  $i^{th}$  row with elements of vector  $x$ , and add them up.

Well it turns out to compute this vector "y", the process is to get  $y_i$ , multiply "A's" "i'th" row with the elements of the vector "x" and add them up. So here's what I mean. In order to get the first element of "y", that first number--whatever that turns out to be--we're gonna take the first row of the matrix "A" and multiply them one at a time with the elements of this vector "x". So I take this first number multiply it by this first number. Then take the second number multiply it by this second number. Take this third number whatever that is, multiply it the third number and so on until you get to the end. And I'm gonna add up the results of these products and the result of adding that up is going to give us this first element of "y". Then when we want to get the second element of "y", let's say this element. The way we do that is we take the second row of A and we repeat the whole thing. So we take the second row of A, and multiply it elements-wise, so the elements of x and add up the results of the products and that would give me the second element of y. And you keep going to get and we are going to take the third row of A, multiply element wise with the vector x, sum up the results and then I get the third element and so on, until I get down to the last row, like so, okay? So that's the procedure.

Let's do one more example. Here's the example:

**Example**

$$\begin{array}{c}
 \boxed{\begin{bmatrix} 1 & 2 & 1 & 5 \\ 0 & 3 & 0 & 4 \\ -1 & -2 & 0 & 0 \end{bmatrix}} \\
 \quad \boxed{3 \times 4}
 \end{array}
 \quad \downarrow
 \quad
 \begin{bmatrix} 1 \\ 3 \\ 2 \\ 1 \end{bmatrix}_{4 \times 1} = \begin{bmatrix} 14 \\ 13 \\ -7 \end{bmatrix}_{3 \times 1} = \begin{bmatrix} 14 \\ 13 \\ -7 \end{bmatrix}$$

$$1 \times 1 + 2 \times 3 + 1 \times 2 + 5 \times 1 = 14$$

$$0 \times 1 + 3 \times 3 + 0 \times 2 + 4 \times 1 = 13$$

$$-1 \times 1 + (-2) \times 3 + 0 \times 2 + 0 \times 1 = -7$$

So let's look at the dimensions. Here, this is a three by four dimensional matrix. This is a four-dimensional vector, or a  $4 \times 1$  matrix, and so the result of this, the result of this product is going to be a three-dimensional vector. Right, you know, the vector, with room for three elements. Let's do the, let's carry out the products. So for the first element, I'm going to take these four numbers and multiply them with the vector  $X$ . So I have  $1 \times 1$ , plus  $2 \times 3$ , plus  $1 \times 2$ , plus  $5 \times 1$ , which is equal to - that's  $1+6$ , plus  $2+6$ , which gives me 14. And then for the second element, I'm going to take this row now and multiply it with this vector  $(0 \times 1) + 3$ . All right, so  $0 \times 1 + 3 \times 3$  plus  $0 \times 2$  plus  $4 \times 1$ , which is equal to, let's see that's  $9+4$ , which is 13. And finally, for the last element, I'm going to take this last row, so I have minus one times one. You have minus two, or really there's a plus next to a two I guess. Times three plus zero times two plus zero times one, and so that's going to be minus one minus six, which is going to make this seven, and so that's vector seven. Okay? So my final answer is this vector fourteen, just to write to that without the colors, fourteen, and thirteen, negative seven. And as promised, the result here is a three by one matrix. So that's how you multiply a matrix and a vector.

I know that a lot just happened on this slide, so if you're not quite sure where all these numbers went, you know, feel free to pause the video you know, and so take a slow careful look at this big calculation that we just did and try to make sure that you understand the steps of what just happened to get us these numbers, fourteen, thirteen and eleven.

Finally, let me show you a **neat trick**.

House sizes:

- 2104
- 1416
- 1534
- 852

Matrix  $\begin{bmatrix} 1 & 2104 \\ 1 & 1416 \\ 1 & 1534 \\ 1 & 852 \end{bmatrix}$   $\begin{matrix} 4 \times 2 \\ \downarrow \end{matrix}$

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

$h_{\theta}(x)$   $\begin{matrix} 2 \times 1 \\ \downarrow \end{matrix}$  Vector  $\begin{bmatrix} -40 \\ 0.25 \end{bmatrix}$

$= \begin{bmatrix} -40 \times 1 + 0.25 \times 2104 \\ -40 \times 1 + 0.25 \times 1416 \\ -40 \times 1 + 0.25 \times 1534 \\ -40 \times 1 + 0.25 \times 852 \end{bmatrix}$   $\begin{matrix} 4 \times 1 \text{ matrix} \\ \uparrow \end{matrix}$

$\boxed{\text{prediction}} = \text{Data Matrix} \times \text{Parameters.}$   $\begin{matrix} 4 \times 1 \\ \uparrow \end{matrix}$

for  $i = 1: 1000$ ,  
 $\text{prediction}(i) = \dots$

Let's say we have a set of four houses so 4 houses with 4 sizes like these. And let's say I have a hypotheses for predicting what is the price of a house, and let's say I want to compute, you know,  $h(x)$  for each of my 4 houses here. It turns out there's a neat way of posing this, applying this hypothesis to all of my houses at the same time. It turns out there's a neat way to pose this as a

Matrix Vector multiplication. So, here's how I'm going to do it. I am going to construct a matrix as follows. My matrix is going to be 1 1 1 1 times, and I'm going to write down the sizes of my four houses here and I'm going to construct a vector as well. And my vector is going to this vector of two elements, that's minus 40 and 0.25. That's these two co-efficients; theta 0 and theta 1. And what I am going to do is to take matrix and that vector and multiply them together, that times is that multiplication symbol. So what do I get? Well this is a four by two matrix. This is a two by one matrix. So the outcome is going to be a four by one vector, all right. So, let me, so this is going to be a 4 by 1 matrix is the outcome or really a four dimensional vector, so let me write it as one of my four elements in my four real numbers here. Now it turns out the first element of this result, the way I am going to get that is, I am going to take this and multiply it by the vector. And so this is going to be  $-40 \times 1 + 0.25 \times 2104$ . By the way, on the earlier slides I was writing  $1 \times -40$  and  $2104 \times 0.25$ , but the order doesn't matter, right?  $-40 \times 1$  is the same as  $1 \times -40$ . And this first element, of course, is  $h(2104)$ . So it's really the predicted price of my first house.

Well, how about the second element? Hope you can see where I am going to get the second element. Right? I'm gonna take this and multiply it by my vector. And so that's gonna be  $-40 \times 1 + 0.25 \times 1416$ . And so this is going to be  $h(1416)$ . Right? And so on for the third and the fourth elements of this 4 x 1 vector. And just there, right? This thing here that I just drew the green box around, that's a real number, OK? That's a single real number, and this thing here that I drew the magenta box around--the purple, magenta color box around--that's a real number, right? And so this thing on the right--this thing on the right overall, this is a 4 by 1 dimensional matrix, is a 4 dimensional vector. And, the neat thing about this is that when you're actually implementing this

in software--so when you have four houses and when you want to use your hypothesis to predict the prices, predict the price "Y" of all of these four houses.

What this means is that, you know, you can write this in one line of code. When we talk about octave and program languages later, you can actually, you'll actually write this in one line of code. You write **prediction equals my, you know, data matrix times parameters, right?** Where data matrix is this thing here, and parameters is this thing here, and this times is a matrix vector multiplication. And if you just do this then this variable prediction - sorry for my bad handwriting - then just implement this one line of code assuming you have an appropriate library to do matrix vector multiplication. If you just do this, then prediction becomes this 4 by 1 dimensional vector, on the right, that just gives you all the predicted prices.

And your alternative to doing this as a matrix vector multiplication would be to write something like, you know, for I equals 1 to 4 (note: a for loop in a programming language), right? And you have say a thousand houses it would be for I equals 1 to a thousand or whatever. And then you have to write a prediction, you know, if I equals. and then do a bunch more work over there and it turns out that When you have a large number of houses, if you're trying to predict the prices of not just four but maybe of a thousand houses then it turns out that when you implement this in the computer, implementing it like this, in any of the various languages. This is not only true for Octave, but for Supra Server Java or Python, other high-level, other languages as well.

It turns out, that, by writing code in this style on the left, it allows you to not only simplify the code, because, now, you're just writing one line of code rather than the form of a bunch of things inside. But, for subtle reasons, that we will see later, it turns out to be much more computationally efficient to make predictions on all of the prices of all of your houses doing it the way on the left than the way on the right than if you were to write your own formula.

I'll say more about this later when we talk about vectorization, but, so, by posing a prediction this way, you get not only a simpler piece of code, but a more efficient one. So, that's it for matrix vector multiplication and we'll make good use of these sorts of operations as we develop the linear regression and other models further.

But, **in the next video** we're going to take this and generalize this to the case of **matrix matrix multiplication.**

## Matrix Matrix Multiplication

In this video we'll talk about **matrix-matrix multiplication**, or how to multiply two matrices together.

When we talk about the method in linear regression for how to solve for the parameters theta 0 and theta 1 all in one shot, without needing an iterative algorithm like gradient descent. When we talk about that algorithm, it turns out that matrix-matrix multiplication is one of the key steps that you need to know.

So let's, as usual, start with an example. Let's say I have two matrices and I want to multiply them together. Let me again just run through this example and then I'll tell you a little bit of what happened.

### Example

$$\begin{array}{c}
 \begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 5 \end{bmatrix} = \begin{bmatrix} 11 \\ 9 \\ 14 \end{bmatrix} \\
 \text{---} \\
 \begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 5 \end{bmatrix} = \begin{bmatrix} 11 \\ 9 \end{bmatrix} \\
 \text{---} \\
 \begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 10 \\ 14 \end{bmatrix}
 \end{array}$$

The diagram illustrates the step-by-step calculation of matrix multiplication. It shows three rows of calculations. The first row shows the multiplication of the 2x3 matrix by the column vector [1, 0, 5] resulting in the column vector [11, 9, 14]. The second row shows the multiplication of the same matrix by the column vector [1, 0, 5] resulting in the row vector [11, 9]. The third row shows the multiplication of the same matrix by the column vector [3, 1, 2] resulting in the row vector [10, 14]. Handwritten annotations include circled '2x3' and '3x2' under the first row, and circled '2x2' under the second row. Arrows point from the circled numbers to the dimensions of the vectors in the first row.

So the first thing I'm gonna do is I'm going to pull out the first column of this matrix on the right. And I'm going to take this matrix on the left and multiply it by a vector that is just this first column. And it turns out, if I do that, I'm going to get the vector 11, 9. So this is the same **matrix-vector multiplication** as you saw in the last video. I worked this out in advance, so I know it's 11, 9.

And then the second thing I want to do is I'm going to pull out the second column of this matrix on the right. And I'm then going to take this matrix on the left, so take that matrix, and multiply it by that second column on the right. So again, this is a **matrix-vector multiplication** step which you saw from the previous video. And it turns out that if you multiply this matrix and this vector you get 10, 14. And by the way, if you want to practice your matrix-vector multiplication, feel free to pause the video and check this product yourself.

Then I'm just gonna take these two results and put them together, and that'll be my answer. So it turns out the outcome of this product is gonna be a two by two matrix. And the way I'm gonna fill in this matrix is just by taking my elements 11, 9, and plugging them here. And taking 10, 14 and plugging them into the second column, okay?

So that was the mechanics of how to multiply a matrix by another matrix. **You basically look at the second matrix one column at a time and you assemble the answers.** And again, we'll step through this much more carefully in a second. But I just want to point out also, this first example is a 2x3 matrix. Multiply that by a 3x2 matrix, and the outcome of this product turns out to be a 2x2 matrix. And again, we'll see in a second why this was the case.

All right, that was the mechanics of the calculation. Let's actually look at the details and look at what exactly happened. Here are the details. I have a matrix A and I want to multiply that with a matrix B and the result will be some new matrix C. **It turns out you can only multiply together matrices whose dimensions match.**

So A is an  $m \times n$  matrix, so m rows, n columns. And we multiply with an  $n \times o$  matrix. And it turns out this n here must match this n here. So the number of columns in the first matrix must equal to the number of rows in the second matrix. And the result of this product will be a  $m \times o$  matrix, like the matrix C here.

### Details:

$$\underline{A} \quad \times \quad \underline{B} \quad = \quad \underline{C}$$

$$\left[ \quad \right] \times \left[ \quad \right] = \left[ \quad \right]$$

$\boxed{m \times n}$  matrix  
(m rows,  
n columns)

$\boxed{n \times o}$  matrix  
(n rows,  
o columns)

$\boxed{m \times o}$  matrix

$\boxed{o = 1}$

And in [the previous video](#) everything we did corresponded to the special case of o being equal to 1. That was to the case of **B being a vector**.

But **now** we're gonna deal with the **case of values of o larger than 1**. So here's how you multiply together the two matrices. What I'm going to do is I'm going to take the first column of B and treat that as a vector, and multiply the matrix A by the first column of B. And the result of that will be a  $n \times 1$  vector, and I'm gonna put that over here.

### Details:

$$\underline{A} \quad \times \quad \underline{B} \quad = \quad \underline{C}$$

$$\left[ \quad \right] \times \left[ \begin{array}{c|c|c|c} \text{green} & \text{pink} & \text{red} & \dots \\ \hline \end{array} \right] = \left[ \begin{array}{c|c|c|c} \text{green} & \text{pink} & \text{red} & \dots \\ \hline \end{array} \right]$$

$\boxed{m \times n}$  matrix  
(m rows,  
n columns)

$\boxed{n \times o}$  matrix  
(n rows,  
o columns)

$\boxed{m \times o}$  matrix

~~$\otimes$~~

The  $i^{th}$  column of the matrix  $C$  is obtained by multiplying  $A$  with the  $i^{th}$  column of  $B$ . (for  $i = 1, 2, \dots, o$ )

Then I'm gonna take the second column of B, right? So this is another n by 1 vector. So this column here, this is n by 1. It's an n-dimensional vector. Gonna multiply this matrix with this n by 1 vector. The result will be a m-dimensional vector, which we'll put there, and so on.

And then I'm gonna take the third column, multiply it by this matrix. I get a m-dimensional vector. And so on, until you get to the last column. The matrix times the last column gives you the last column of C. Just to say that again, the ith column of the matrix C is obtained by taking the matrix A and multiplying the matrix A with the ith column of the matrix B for the values of i = 1, 2, up through o. So this is just a summary of what we did up there in order to compute the matrix C.

Let's look at just **one more example**.

Let's say I want to **multiply together these two matrices**.

### Example

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

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

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

So what I'm going to do is first pull out the first column of my second matrix. That was my matrix B on the previous slide and I therefore have this matrix times that vector. And so, oh, let's do this calculation quickly. This is going to be equal to the 1, 3 x 0, 3, so that gives 1 x 0 + 3 x 3. And the second element is going to be 2, 5 x 0, 3, so that's gonna be 2 x 0 + 5 x 3. And that is 9, 15. Oh, actually let me write that in green.

So this is 9, 15. And then next I'm going to pull out the second column of this and do the corresponding calculations. So that's this matrix times this vector 1, 2. Let's also do this quickly, so that's 1 x 1 + 3 x 2, so that was that row. And let's do the other one. So let's see, that gives me 2 x 1 + 5 x 2 and so that is going to be equal to, lets see, 1 x 1 + 3 x 1 is 4 and 2 x 1 + 5 x 2 is 12. So now I have these two and so my outcome, the product of these two matrices, is going to be this goes here and this goes here. So I get 9, 15 and 4, 12. [Note: Error in video: Second column should be 7, 12] And you may notice also that the result of multiplying a 2x2 matrix with another 2x2 matrix, the resulting dimension is going to be that first 2 times that second 2. So the result is itself also a 2x2 matrix.

Finally, let me show you **one more neat trick** that you can do with matrix-matrix multiplication. Let's say, as before, that we have **four houses** whose prices we wanna predict. Only now, we have **three competing hypotheses** shown here on the right. So if you want to **apply all three**

**competing hypotheses to all four of your houses**, it turns out you can do that very efficiently using a matrix-matrix multiplication. So here on the left is my usual matrix, same as from the last video where these values are my housing sizes and I've put 1s here on the left as well.

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 $\begin{bmatrix} 1 & 2104 \\ 1 & 1416 \\ 1 & 1534 \\ 1 & 852 \end{bmatrix}$	×	Matrix $\begin{bmatrix} -40 \\ 200 \\ -150 \\ 0.25 \end{bmatrix}$
---	---	--

$=$

$\uparrow$ Prediction of 1st h <sub>θ</sub>	$\uparrow$ Predictions of 2nd h <sub>θ</sub>	$\begin{bmatrix} 486 & 410 & 692 \\ 314 & 342 & 416 \\ 344 & 353 & 464 \\ 173 & 285 & 191 \end{bmatrix}$
---	--	--

And what I am going to do is construct another matrix where here, the first column is this -40 and 0.25 and the second column is this 200, 0.1 and so on. And it turns out that if you multiply these two matrices, what you find is that this first column, I'll draw that in blue. Well, how do you get this first column? Our procedure for matrix-matrix multiplication is, the way you get this first column is you take this matrix and you multiply it by this first column. And we saw in the previous video that this is exactly the predicted housing prices of the first hypothesis, right, of this first hypothesis here. And how about the second column? Well, [INAUDIBLE] second column. The way you get the second column is, well, you take this matrix and you multiply it by this second column. And so the second column turns out to be the predictions of the second hypothesis up there, and similarly for the third column. And so I didn't step through all the details, but hopefully you can just feel free to pause the video and check the math yourself and check that what I just claimed really is true. But it turns out that by constructing these two matrices, what you can therefore do is very quickly apply all 3 hypotheses to all 4 house sizes to get all 12 predicted prices output by your 3 hypotheses on your 4 houses. **So with just one matrix multiplication step you managed to make 12 predictions**. And even better, it turns out that in order to do that matrix multiplication, there are lots of good linear algebra libraries in order to do this multiplication step for you. And so pretty much any reasonable programming language that you might be using. Certainly all the top ten most popular programming languages will have great linear algebra libraries.

And there'll be good linear algebra libraries that are highly optimized in order to do that matrix-matrix multiplication very efficiently. Including taking advantage of any sort of parallel computation that your computer may be capable of, whether your computer has multiple cores or multiple processors. Or within a processor sometimes there's parallelism as well called SIMD parallelism that your computer can take care of. And there are very good free libraries that you

can use to do this matrix-matrix multiplication very efficiently, so that you can very efficiently make lots of predictions with lots of hypotheses.

## Matrix Multiplication Properties

Matrix multiplication is really useful, since you can pack a lot of computation into just one matrix multiplication operation. But you should be careful of how you use them.

### Not Commutative

In this video, I wanna tell you about a few properties of matrix multiplication. When working with just real numbers or when working with scalars, multiplication is commutative. And what I mean by that is that if you take 3 times 5, that is equal to 5 times 3. And the ordering of this multiplication doesn't matter. And this is called the commutative property of multiplication of real numbers. It turns out this property, they can reverse the order in which you multiply things. This is not true for matrix multiplication.

$$3 \times 5 = 5 \times 3 \quad \text{"Commutative"}$$

Let  $A$  and  $B$  be matrices. Then in general,

$A \times B \neq B \times A$ . (not commutative.)

So concretely, if  $A$  and  $B$  are matrices. Then in general,  $A$  times  $B$  is not equal to  $B$  times  $A$ . So, just be careful of that. Its not okay to arbitrarily reverse the order in which you multiply matrices. Matrix multiplication in not commutative, is the fancy way of saying it.

As a concrete example, here are two matrices. This matrix  $\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$  times  $\begin{bmatrix} 0 & 0 \\ 2 & 0 \end{bmatrix}$  is  $\begin{bmatrix} 2 & 0 \\ 0 & 0 \end{bmatrix}$ . If you multiply these two matrices you get this result on the right. Now let's swap around the order of these two matrices. So I'm gonna take this two matrices and just reverse them. It turns out if you multiply these two matrices, you get the second answer on the right. And well clearly, right, these two matrices are not equal to each other.

<p>E.g. <math>\begin{bmatrix} 1 &amp; 1 \\ 0 &amp; 0 \end{bmatrix} \begin{bmatrix} 0 &amp; 0 \\ 2 &amp; 0 \end{bmatrix} = \begin{bmatrix} 2 &amp; 0 \\ 0 &amp; 0 \end{bmatrix}</math></p> <p><del><math>\begin{bmatrix} 0 &amp; 0 \\ 2 &amp; 0 \end{bmatrix} \begin{bmatrix} 1 &amp; 1 \\ 0 &amp; 0 \end{bmatrix} = \begin{bmatrix} 0 &amp; 0 \\ 2 &amp; 2 \end{bmatrix}</math></del></p>	$A \times B$ $m \times n$ $n \times m$ <hr/> $A \times B$ $m \times m$ <hr/> $B \times A$ $n \times n$
---	---

So, in fact, in general if you have a matrix operation like  $A$  times  $B$ , if  $A$  is an  $m$  by  $n$  matrix, and  $B$  is an  $n$  by  $m$  matrix, just as an example. Then, it turns out that the matrix  $A$  times  $B$ , right, is going to be an  $m$  by  $m$  matrix. Whereas the matrix  $B$  times  $A$  is going to be an  $n$  by  $n$  matrix. So

the dimensions don't even match, right? So if  $A \times B$  and  $B \times A$  may not even be the same dimension.

In the example on the left, I have all two by two matrices. So the dimensions were the same, but in general, reversing the order of the matrices can even change the dimension of the outcome. So, **matrix multiplication is not commutative**.

### Associative

Here's the next property I want to talk about. So, when talking about real numbers or scalars, let's say I have  $3 \times 5 \times 2$ . I can either multiply  $5 \times 2$  first. Then I can compute this as  $3 \times 10$ . Or, I can multiply  $3 \times 5$  first, and I can compute this as  $15 \times 2$ . And both of these give you the same answer, right? Both of these is equal to 30. So it doesn't matter whether I multiply  $5 \times 2$  first or whether I multiply  $3 \times 5$  first, because sort of, well,  $3 \times (5 \times 2) = (3 \times 5) \times 2$ . And this is called the associative property of real number multiplication.

$$\begin{array}{ccc} \cancel{3 \times 5 \times 2} & 3 \times (5 \times 2) = (3 \times 5) \times 2 \\ 3 \times 10 = 30 = 15 \times 2 & & \text{"Associative"} \end{array}$$

It turns out that matrix multiplication is associative. So concretely, let's say I have a product of three matrices  $A \times B \times C$ . Then, I can compute this either as  $A \times (B \times C)$  or I can computer this as  $(A \times B) \times C$ , and these will actually give me the same answer. I'm not gonna prove this but you can just take my word for it I guess.

$$\begin{array}{c} A \times (B \times C) \quad \leftarrow \\ (A \times B) \times C \quad \leftarrow \\ A \times B \times C. \end{array}$$

Let  $D = B \times C$ . Compute  $A \times D$ .

Let  $E = A \times B$ . Compute  $E \times C$ .

So just be clear, what I mean by these two cases. Let's look at the first one, right. This first case. What I mean by that is if you actually wanna compute  $A \times B \times C$ . What you can do is you can first compute  $B \times C$ . So that  $D = B \times C$  then compute  $A \times D$ . And so this here is really computing  $A \times B \times C$ . Or, for this second case, you can compute this as, you can set  $E = A \times B$ , then compute  $E \times C$ . And this is then the same as  $A \times B \times C$ , and it turns out that both of these options will give you this guarantee to give you the same answer. And so we say that matrix multiplication thus enjoy the associative property. Okay?

And don't worry about the terminology associative and commutative. That's what it's called, but I'm not really going to use this terminology later in this class, so don't worry about memorizing those terms.

### Identity Matrix

Finally, I want to tell you about the Identity Matrix, which is a special matrix. So let's again make the analogy to what we know of real numbers. When dealing with real numbers or scalar numbers, the number 1, you can think of it as the identity of multiplication. And what I mean by that is that for any number  $z$ ,  $1 \times z = z \times 1 = z$ . And that's just equal to the number  $z$  for any real number  $z$ . So 1 is the identity operation and so it satisfies this equation.

$$1 \text{ is identity. } \boxed{1 \times z = z \times 1 = z} \\ \text{for any } z$$

So it turns out, that this in the space of matrices there's an identity matrix as well and it's usually denoted  $I$  or sometimes we write it as  $I_{n \times n}$  if we want to make it explicit to dimensions.

## Identity Matrix

Denoted  $I$  (or  $I_{n \times n}$ ).

Examples of identity matrices:

$$\begin{bmatrix} 1 \end{bmatrix} \quad 1 \times 1$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad 3 \times 3$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad 4 \times 4$$

Informally:

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

For any matrix  $A$ ,

$$A \cdot \boxed{I} = \boxed{I} \cdot A = A$$

$\overset{m \times n}{\nearrow}$        $\overset{n \times n}{\nearrow}$        $\overset{m \times m}{\nearrow}$        $\overset{m \times n}{\searrow}$

$$I_{n \times n}$$

Note:  
 $AB \neq BA$  in general  
 $AI = IA$  ✓

So  $I_{n \times n}$  is the  $n \times n$  identity matrix. And so that's a different identity matrix for each dimension  $n$ . And here are few examples. Here's the  $2 \times 2$  identity matrix, here's the  $3 \times 3$  identity matrix, here's the  $4 \times 4$  matrix. So the identity matrix has the property that it has ones along the diagonals. All right, and so on. And 0 everywhere else. And so, by the way, the  $1 \times 1$  identity matrix is just a number 1, and so the  $1 \times 1$  matrix with just 1 in it. So it's not a very interesting identity matrix. And informally, when I or others are being sloppy, very often we'll write the identity matrices in fine notation. We'll draw square brackets, just write one one one dot dot dot dot one, and then we'll maybe somewhat sloppily write a bunch of zeros there. And these zeroes on the, this big zero and this big zero, that's meant to denote that this matrix is zero everywhere except for the diagonal. So this is just how I might swap the identity matrix. And it turns out that the identity matrix has its property that for any matrix  $A$ ,  $A$  times identity equals  $I$  times  $A$  equals  $A$  so that's a lot like this equation that we have up here. Right? So  $1$  times  $z$  equals  $z$  times  $1$  equals  $z$  itself. So  $I$  times  $A$  equals  $A$  times  $I$  equals  $A$ . Just to make sure we have the dimensions right. So if  $A$  is an  $m$  by  $n$  matrix, then this identity matrix here, that's an  $n$  by  $n$  identity matrix. And if is and by then, then this identity matrix, right? For matrix multiplication to make sense, that has to be an  $m$  by  $m$  matrix. Because this  $m$  has to match up that  $m$ , and in either case, the outcome of this process is you get back the matrix  $A$  which is  $m$  by  $n$ . So

whenever we write the identity matrix  $I$ , you know, very often the dimension, right, will be implicit from the context. So these two  $I$ 's, they're actually different dimension matrices. One may be  $n$  by  $n$ , the other is  $n$  by  $m$ . But when we want to make the dimension of the matrix explicit, then sometimes we'll write to this  $I$  subscript  $n$  by  $n$ , kind of like we had up here. But very often, the dimension will be implicit.

Finally, I just wanna point out that earlier I said that  $AB$  is not, in general, equal to  $BA$ . Right? For most matrices  $A$  and  $B$ , this is not true. But when  $B$  is the identity matrix, this does hold true, that  $A$  times the identity matrix does indeed equal to identity times  $A$  is just that you know this is not true for other matrices  $B$  in general.

So, that's it for the properties of matrix multiplication and special matrices like the identity matrix I want to tell you about.

In the next and final video on our linear algebra review, I'm going to quickly tell you about a couple of **special matrix operations** and after that everything you need to know about linear algebra for this class.

## Inverse and Transpose

In this video, I want to tell you about a couple of special matrix operations, called the **matrix inverse** and the **matrix transpose** operation.

Let's start by talking about **matrix inverse**, and as usual we'll start by thinking about how it relates to real numbers.

In the last video, I said that the number one plays the role of the identity in the space of real numbers because one times anything is equal to itself. It turns out that real numbers have this property that every number has an inverse, that each number has an inverse, for example, given the number 3, there exists some number, which happens to be 3 inverse so that that number times gives you back the identity element one. And so to me, inverse of course this is just one third.

$$\begin{array}{lll} \underline{1 = \text{"identity."}} & 3 \left[ \frac{(3^{-1})}{\frac{1}{3}} \right] = 1 & 12 \times \frac{1}{12} = 1 \\ & \text{Not all numbers have an inverse.} & \end{array}$$

And given some other number, maybe 12 there is some number which is the inverse of 12 written as twelve to the minus one, or really this is just one twelve. So that when you multiply these two things together, the product is equal to the identity element one again.

Now it turns out that in the space of real numbers, not everything has an inverse. For example the number zero does not have an inverse, right? Because zero's a zero inverse, one over zero that's undefined. Like this one over zero is not well defined.

### Inverse of Matrix

And what we want to do, in the rest of this slide, is **figure out what does it mean to compute the inverse of a matrix.**

Here's the idea: If  $A$  is a  $n$  by  $n$  matrix, and it has an inverse, I will say a bit more about that later, then the inverse is going to be written  $A$  to the minus one and  $A$  times this inverse,  $A$  to the minus one, is going to equal to  $A$  inverse times  $A$ , is going to give us back the identity matrix. Okay?

**Only matrices that are  $m$  by  $m$  for some value of  $m$  having inverse.** So, a matrix is  $M$  by  $M$ , this is **also called a square matrix**, and it's called square because the number of rows is equal to the number of columns. Right and it turns out **only square matrices have inverses**, so  $A$  is a square matrix, is  $m$  by  $m$ , for inverse this satisfies the equation over here.

Let's look at a **concrete example**.

So let's say I have a matrix, three, four, two, and sixteen.

**Matrix inverse:** *( $\leftarrow$  Square matrix  
 $\leftarrow$  #rows = #columns)*  $A^{-1}$

If  $A$  is an  $m \times m$  matrix, and if it has an inverse,

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

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

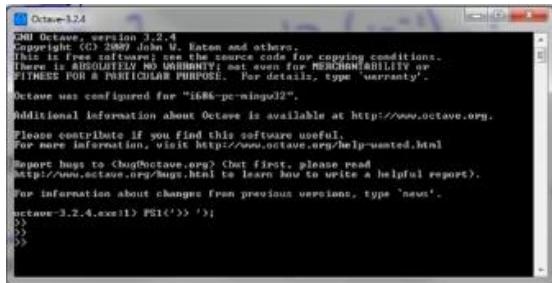
$A \quad A^{-1} \quad A^{-1}A$

So this is a two by two matrix, so it's a square matrix and so this may just could have an inverse and it turns out that I happen to know the inverse of this matrix is zero point four, minus zero point one, minus zero point zero five, zero point zero seven five. **And if I take this matrix and multiply these together it turns out what I get is the two by two identity matrix,  $I$ , this is  $I$  two by two. Okay? And so on this slide, you know this matrix is the matrix  $A$ , and this matrix is the matrix  $A$ -inverse. And it turns out if that you are computing  $A$  times  $A$ -inverse, it turns out if you compute  $A$ -inverse times  $A$  you get back the identity matrix.** So how did I find this inverse or how did I come up with this inverse over here?

It turns out that **sometimes you can compute inverses by hand but almost no one does that these days.** And it turns out there is very good numerical software for taking a matrix and computing its inverse. So again, this is one of those things where there are lots of open source libraries that you can link to from any of the popular programming languages to compute inverses of matrices.

Let me show you a quick example.

How I actually computed this inverse, and what I did was I used software called **Octave**. So let me bring that up. We will see a lot about Octave later.



Let me just quickly show you an example. Set my matrix A to be equal to that matrix on the left, type  $\begin{bmatrix} 3 & 4 \\ 2 & 16 \end{bmatrix}$ , so that's my matrix A right.

```

Octave-3.2.4
>> A = [ 3 4; 2 16 ]
A =
    3    4
    2   16
>> pinv(A)
ans =
    0.400000  -0.100000
   -0.050000   0.075000
>> inverseOfA = pinv(A)
inverseOfA =
    0.400000  -0.100000
   -0.050000   0.075000
>> A * inverseOfA
ans =
    1.0000e+000  -5.5511e-017
   -2.2204e-016   1.0000e+000
>>

```

This is matrix  $\begin{bmatrix} 3 & 4 \\ 2 & 16 \end{bmatrix}$  that I have down here on the left. And, the software lets me compute the inverse of A very easily. It's like **pinv(A)** equals this. And so, this is right, this matrix here 0.4, --0.1, and so on.

This has given me the numerical solution to what is the **inverse of A**. So let me just write, inverseOfA equals **pinv(A)** over that I can now just verify that A times A inverse the identity is, type A times the inverse of A and the result of that is this matrix and this is 1 1 on the diagonal and essentially ten to the minus seventeen, ten to the minus sixteen, so up to numerical precision, up to a little bit of round off error that my computer had in finding optimal matrices and these numbers off the diagonals are essentially zero so **A times the inverse is essentially the identity matrix.**

### Singular or Degenerate Matrix

Can also verify the **inverse of A times A is also equal to the identity**, ones on the diagonals and values that are essentially zero except for a little bit of round off error on the off diagonals. If a definition that the inverse of a matrix is, I had this caveat first it must always be a square matrix, it had this caveat, that if A has an inverse, exactly what matrices have an inverse is beyond the scope of this linear algebra for review that one intuition you might take away that just as the number zero doesn't have an inverse.

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

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

Andrew Ng

It turns out that if  $A$  is say the matrix of all zeros, then this matrix  $A$  also does not have an inverse because there's no matrix there's no  $A$  inverse matrix so that this matrix times some other matrix will give you the identity matrix so this matrix of all zeros, and there are a few other matrices with properties similar to this. That also don't have an inverse.

But it turns out that in this review I don't want to go too deeply into what it means matrix have an inverse but it turns out for our machine learning application this shouldn't be an issue or more precisely for the learning algorithms where this may be an issue to namely whether or not an inverse matrix appears and I will tell when we get to those learning algorithms just what it means for an algorithm to have or not have an inverse and how to fix it in case working with matrices that don't have inverses. **But the intuition if you want is that you can think of matrices as not have an inverse that is somehow too close to zero in some sense.** So, just to wrap up the terminology, matrix that don't have an inverse Sometimes called a **singular matrix or degenerate matrix** and so this matrix over here is an example zero zero zero matrix is an example of a matrix that is singular, or a matrix that is degenerate.

### Transpose

Finally, the last special matrix operation I want to tell you about is to do matrix transpose.

So suppose I have matrix  $A$ , if I compute the transpose of  $A$ , that's what I get here on the right. This is a transpose which is written and  $A$  superscript T, and the way you compute the transpose of a matrix is as follows. To get a transpose I am going to first take the first row of  $A$  one to zero. That becomes this first column of this transpose. And then I'm going to take the second row of  $A$ , 3 5 9, and that becomes the second column. of the matrix  $A$  transpose. And another way of thinking about how the computer transposes is as if you're taking this sort of 45 degree axis and you are mirroring or you are flipping the matrix along that 45 degree axis. so here's the more formal definition of a matrix transpose.

## Matrix Transpose

Example:

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

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

Let  $A$  be an  $m \times n$  matrix, and let  $B = A^T$ .

Then  $B$  is an  $n \times m$  matrix, and

$$\underline{B}_{ij} = \underline{A}_{ji}.$$

$$\underline{B}_{12} = \underline{A}_{21} = 2$$

$$\underline{B}_{32} = 9 \quad \underline{A}_{23} = 9.$$

Let's say  $A$  is a  $m$  by  $n$  matrix. And let's let  $B$  equal  $A$  transpose and so  $B$  equals  $A$  transpose like so. Then  $B$  is going to be a  $n$  by  $m$  matrix with the **dimensions reversed** so here we have a  $2 \times 3$  matrix. And so the transpose becomes a  $3 \times 2$  matrix, and moreover, the  $B_{ij}$  is equal to  $A_{ji}$ . So the  $ij$  element of this matrix  $B$  is going to be the  $ji$  element of that earlier matrix  $A$ . So for example,  $B_{12}$  is going to be equal to, look at this matrix  $B_{12}$  is going to be equal to this element 3, 1st row, 2nd column. And that equal to this, which is a two one, second row first column, right, which is equal to 3 and some other example  $B_{32}$ , right, that's  $B_{32}$  is this element 9, and that's equal to  $A_{23}$  which is this element up here, 9. And so that wraps up the definition of what it means to take the transpose of a matrix and that in fact concludes our linear algebra review.

So by now hopefully you know **how to add and subtract matrices as well as multiply them** and you also know how, what are the definitions of the **inverses and transposes of a matrix** **and these are the main operations used in linear algebra** for this course.

In case this is the first time you are seeing this material. I know this was a lot of linear algebra material all presented very quickly and it's a lot to absorb but if you there's no need to memorize all the definitions we just went through and if you download the copy of either these slides or of the lecture notes from the course website. and use either the slides or the lecture notes as a reference then you can always refer back to the definitions and to figure out what are these matrix multiplications, transposes and so on definitions. And the lecture notes on the course website also has pointers to additional resources linear algebra which you can use to learn more about linear algebra by yourself.

And next with these new tools. We'll be able in the **next few videos to develop more powerful forms of linear regression** that can view of a lot more data, a lot more features, a lot more training examples and later on after the new regression we'll actually continue using these linear algebra tools to derive more powerful learning algorithms as well.

# Linear Regression with Multiple Variables

## Multiple Features

In this video we will start to talk about **a new version of linear regression** that's more powerful. One that **works with multiple variables** or **with multiple features**. Here's what I mean.

In the original version of linear regression that we developed, we have a single feature  $x$ , the size of the house, and we wanted to use that to predict  $y$ , the price of the house and this was our form of our hypothesis.

Size (feet <sup>2</sup> )	Price (\$1000)
$\rightarrow x$	$y \rightarrow$
2104	460
1416	232
1534	315
852	178
...	...

$$h_{\theta}(x) = \theta_0 + \theta_1 x$$

But now imagine, what if we had not only the size of the house as a feature or as a variable of which to try to predict the price, but that we also knew the number of bedrooms, the number of floors and the age of the home and years. It seems like this would give us a lot more information with which to predict the price.

### Multiple features (variables).

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

To introduce a little bit of notation, we sort of started to talk about this earlier, I'm going to use the variables  $X$  subscript 1  $X$  subscript 2 and so on to denote my, in this case, four features and I'm going to continue to use  $Y$  to denote the variable, the output variable price that we're trying to predict.

Let's introduce a little bit more notation. Now that we have four features I'm going to use **lowercase "n" to denote the number of features**. So in this example we have  $n$  equals 4 because we have, you know, one, two, three, four features. And " $n$ " is different from our earlier

notation where we were using "m" to denote the number of examples. So if you have 47 rows "m" is the number of rows on this table or the number of training examples.

### Multiple features (variables).

<u>Size (feet<sup>2</sup>)</u>	<u>Number of bedrooms</u>	<u>Number of floors</u>	<u>Age of home (years)</u>	<u>Price (\$1000)</u>
$x_1$	$x_2$	$x_3$	$x_4$	$y$
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
...	...	...	...	...
<u>Notation:</u>	<u><math>x^{(2)}</math></u>	<u><math>x^{(3)}</math></u>	<u><math>x^{(4)}</math></u>	

→  $n$  = number of features  $n=4$   
 →  $x^{(i)}$  = input (features) of  $i^{\text{th}}$  training example.  
 →  $x_j^{(i)}$  = value of feature  $j$  in  $i^{\text{th}}$  training example.

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

So I'm also going to use **X superscript "i"** to denote the input features of the "**i<sup>th</sup>**" training example. As a concrete example let say **x<sup>2</sup> is going to be a vector of the features for my second training example**. And so **x<sup>2</sup>** here is going to be a vector 1416, 3, 2, 40 since those are my four features that I have to try to predict the price of the second house. So, in this notation, the **superscript 2 here, that's an index into my training set**. This is not X to the power of 2. Instead, this is, you know, an index that says look at the second row of this table. This refers to my second training example. With this notation **x<sup>2</sup>** is a four dimensional vector. In fact, more generally, this is an **n-dimensional feature vector**. With this notation, **x<sup>2</sup>** is now a vector and so, I'm going to use also **x<sup>i</sup> subscript j** to denote the value of the **j<sup>th</sup>** feature, of feature number **j** and in the **i<sup>th</sup>** training example. So concretely **x<sup>2</sup><sub>3</sub>**, will refer to feature number three in this vector which is equal to 2. Right? That was a 3 over there, just fix my handwriting. So **x<sup>2</sup><sub>3</sub>** is going to be equal to 2.

Now that we have multiple features, let's talk about what the form of our hypothesis should be. Previously this was the form of our hypothesis, where x was our single feature, but now that we have multiple features, we aren't going to use the simple representation any more.

### Hypothesis:

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

Instead, a form of the hypothesis in linear regression is going to be this, can be  $\theta_0$  plus  $\theta_1 x_1$  plus  $\theta_2 x_2$  plus  $\theta_3 x_3$  plus  $\theta_4 x_4$ . And if we have N features then rather than summing up over our four features, we would have a sum over our N features.

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

Concretely for a particular setting of our parameters we may have  $h(x)$  equals  $80 + 0.1 x_1 + 0.01 x_2 + 3 x_3 - 2 x_4$ .

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

↑      ↑      ↑      ↑  
                age

This would be one example of a hypothesis, and you remember a hypothesis is trying to predict the price of the house in thousands of dollars, just saying that, you know, the base price of a house is maybe 80,000 plus another 0 point 1, so that's an extra, what, hundred dollars per square feet, yeah, plus the price goes up a little bit for each additional floor that the house has.  $x_2$  is the number of floors, and it goes up further for each additional bedroom the house has, because  $x_3$  was the number of bedrooms, and the price goes down a little bit with each additional age of the house. With each additional year of the age of the house.

Here's the form of a hypothesis rewritten on the slide. And what I'm gonna do is introduce a little bit of notation to simplify this equation.

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

**For convenience** of notation, let me define  $x_0$  to be equals one. Concretely, this means that for every example  $i$  I have a feature vector  $\mathbf{x}^{(i)}$ , and  $\mathbf{x}^{(i)}_0$  is going to be equal to 1. You can think of this as **defining an additional zero feature**. So whereas previously I had  $n$  features because  $x_1, x_2$  through  $x_n$ , I'm now defining an additional sort of zero feature vector that **always takes on the value of one**. So now my **feature vector  $X$  becomes this  $N+1$  dimensional vector** that is zero index. So this is now a  $n+1$  dimensional feature vector, but I'm gonna index it from 0 and **I'm also going to think of my parameters as a vector**. So, our parameters here, right that would be our theta zero, theta one, theta two, and so on all the way up to theta  $n$ , we're going to gather them up into a **parameter vector** written theta 0, theta 1, theta 2, and so on, down to theta  $n$ . This is another zero index vector. It's of index signed from zero. That is another  $n+1$  dimensional vector.

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

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

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

So, my hypothesis can now be written  $\theta_0 x_0$  plus  $\theta_1 x_1$  plus dot dot dot up to  $\theta_n x_n$ . And this equation is the same as this on top because, you know,  $x_0$  is equal to one.

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

$\downarrow = 1$

Underneath thing is now **I can take this form of the hypothesis and write this as theta transpose x**, depending on how familiar you are with inner products of vectors, if you write what theta transpose x is, what theta transpose x is that this is theta zero, theta one, up to theta N.

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

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

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n+1}$$

$$\Theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} \in \mathbb{R}^{n+1}$$

$$h_{\theta}(x) = \underline{\theta_0x_0 + \theta_1x_1 + \cdots + \theta_nx_n} = \underline{\Theta^T x}$$

$\Theta^T$   
 $(n+1) \times 1$  matrix  
 $\Theta^T x$

### Multivariate linear regression. ←

So this thing here is **theta transpose** and this is actually a **one by N plus one matrix**. It's also **called a row vector** and you take that and multiply it with the vector X which is X zero, X one, and so on, down to X n. And so, the inner product that is theta transpose X is just equal to this. This gives us a convenient way to write the form of the **hypothesis** as just the **inner product** between our **parameter vector theta** and our **feature vector X**. And it is this little bit of notation, this little extra bit of notation convention that let us write this in this compact form. So that's the form of a hypothesis when we have multiple features.

And, just to give this another name, this is also called **multivariate linear regression**. And the term multivariable that's just maybe a fancy term for saying we have multiple features, or multivariables with which to try to predict the value Y.

## Gradient Descent for Multiple Variables

In the previous video, we talked about the form of the hypothesis for linear regression with multiple features or with multiple variables.

In this video, let's talk about **how to fit the parameters of that hypothesis**. In particular let's talk about **how to use gradient descent for linear regression with multiple features**.

To quickly summarize our notation, this is our formal hypothesis in multivariable linear regression where we've adopted the convention that  $x_0 = 1$ .

Hypothesis:  $h_{\theta}(x) = \theta^T x = \theta_0x_0 + \theta_1x_1 + \theta_2x_2 + \cdots + \theta_nx_n$

The parameters of this model are theta 0 through theta n, but instead of thinking of this as n separate parameters, which is valid, I'm instead going to think of the parameters as **theta** where **theta here is a n+1-dimensional vector**. So I'm just going to think of the parameters of this model as itself being a vector.

Parameters:  $\underline{\theta_0, \theta_1, \dots, \theta_n}$   $\underline{\theta}$   $n+1$ -dimensional vector

Our cost function is J (theta 0 through theta n) which is given by this usual sum of square of error term. But again instead of thinking of J as a function of these n+1 numbers, I'm going to more commonly write **J as just a function of the parameter vector theta** so that theta here is a vector.

Cost function:

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

Here's what gradient descent looks like. We're going to repeatedly update each parameter theta j according to **theta j minus alpha times this derivative term**. And once again we just write this as J of theta, so theta j is updated as theta j minus the **learning rate alpha** times the derivative, a partial derivative of the cost function with respect to the parameter theta j.

Gradient descent:

$$\begin{aligned} & \text{Repeat } \{ \\ & \quad \rightarrow \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \dots, \theta_n) \quad \underline{J(\theta)} \\ & \quad \} \quad \text{(simultaneously update for every } j = 0, \dots, n) \end{aligned}$$

Let's see what this looks like when we implement gradient descent and, in particular, let's go see what that partial derivative term looks like.

Here's what we have for gradient descent for the case of when we had n=1 feature. We had two **separate update rules for the parameters theta0 and theta1**, and hopefully these look familiar to you. And this term here was of course the **partial derivative of the cost function with respect to the parameter of theta0**, and similarly we had a different update rule for the parameter theta1.

### Gradient Descent

Previously (n=1):

$$\begin{aligned} & \text{Repeat } \{ \\ & \quad \rightarrow \theta_0 := \theta_0 - \alpha \underbrace{\frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})}_{\frac{\partial}{\partial \theta_0} J(\theta)} \\ & \quad \rightarrow \theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x^{(i)} \\ & \quad \quad \quad \text{(simultaneously update } \theta_0, \theta_1) \\ & \} \end{aligned}$$

There's one little difference which is that when we previously had only one feature, we would call that feature  $\mathbf{x}^{(i)}$  but now in our new notation we would of course call this  $\mathbf{x}^{(i)}_1$  to denote our one feature. So that was for when we had only one feature.

Let's look at the new algorithm for we have more than one feature, where the number of features  $n$  may be much larger than one.

## Gradient Descent

Previously ( $n=1$ ):

Repeat {

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

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

(simultaneously update  $\theta_0, \theta_1$ )

}

New algorithm ( $n \geq 1$ ):

Repeat {

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

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

}

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

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

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

...

We get this update rule for gradient descent and, maybe for those of you that know calculus, if you take the definition of the cost function and take the partial derivative of the cost function  $J$  with respect to the parameter  $\theta_j$ , you'll find that that partial derivative is exactly that term that I've drawn the blue box around. And if you implement this you will get a working implementation of gradient descent for multivariate linear regression.

The last thing I want to do on this slide is give you a sense of why these new and old algorithms are sort of the same thing or why they're both similar algorithms, or why they're both gradient descent algorithms.

Let's consider a case where we have two features or maybe more than two features, so we have three update rules for the parameters  $\theta_0, \theta_1, \theta_2$  and maybe other values of  $\theta$  as well. If you look at the update rule for  $\theta_0$ , what you find is that this update rule here is the same as the update rule that we had previously for the case of  $n = 1$ . And the reason that they are equivalent is, of course, because in our notational convention we had this  $\mathbf{x}_0^{(1)} = 1$  convention, which is why these two terms that I've drawn the magenta boxes around are equivalent.

Similarly, if you look the update rule for  $\theta_1$ , you find that this term here is equivalent to the term we previously had, or the equation or the update rule we previously had for  $\theta_1$ , where of course we're just using this new notation  $\mathbf{x}_1^{(i)}$  to denote our first feature, and now that we have more than one feature we can have similar update rules for the other parameters like  $\theta_2$  and so on.

There's a lot going on on this slide so I definitely encourage you if you need to pause the video and look at all the math on this slide slowly to make sure you understand everything that's going on here. But if you implement the algorithm written up here then you have a working implementation of linear regression with multiple features.

## Gradient Descent in Practice I - Feature Scaling

In this video and in the video after this one, I wanna tell you about some of the practical tricks for making gradient descent work well.

In this video, I want to tell you about an idea called **feature scaling**.

Here's the idea. If you have a problem where you have multiple features, if you make sure that the features are on a similar scale, by which I mean make sure that the different features take on similar ranges of values, then gradient descents can converge more quickly.

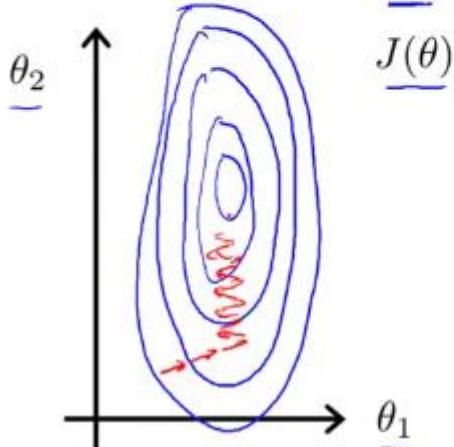
Concretely let's say you have a problem with two features where  $x_1$  is the size of house and takes on values between say 0 to 2000 and  $x_2$  is the number of bedrooms, and maybe that takes on values between 1 and 5.

### Feature Scaling

Idea: Make sure features are on a similar scale.

E.g.  $x_1$  = size (0-2000 feet<sup>2</sup>) ←

$x_2$  = number of bedrooms (1-5) ←



If you plot the contours of the cost function  $J(\theta)$ , then the contours may look like this, where, let's see,  $J(\theta)$ , is a function of parameters  $\theta_0$ ,  $\theta_1$  and  $\theta_2$ . I'm going to ignore  $\theta_0$ , so let's forget about theta 0 and pretend it as a function of only theta 1 and theta 2. But if  $x_1$  can take on, you know, much larger range of values than  $x_2$ , then it turns out that the contours of the cost function  $J(\theta)$  can take on this sort of a very very skewed elliptical shape, except that with the so 2000 to 5 ratio, it can be even more skew. So, this is very, very tall and skinny ellipses, or these very tall skinny ovals, can form the contours of the cost function  $J(\theta)$ . And if you run gradient descents on this sort of a cost function, your gradients may end up taking a long time and can sort of

oscillate back and forth and take a long time before it can finally find its way to the **global minimum.**

In fact, you can imagine if these contours are exaggerated even more when you draw incredibly skinny, tall skinny contours, and it can be even more extreme than that, then, gradient descent just have a much harder time taking its way, meandering around, it can take a long time to find this way to the global minimum.

**In these settings, a useful thing to do is to scale the features.**

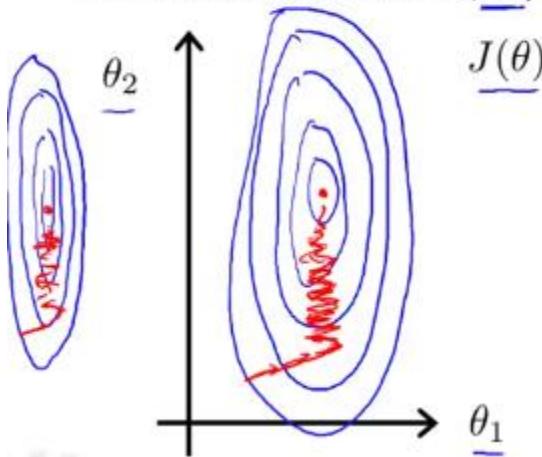
Concretely if you instead define the feature  $x_1$  to be the size of the house divided by two thousand, and define  $x_2$  to be maybe the number of bedrooms divided by five, then the contours of the cost function  $J$  can become much more, much less skewed so the contours may look more like circles. And if you run gradient descent on a cost function like this, then gradient descent, you can show mathematically, you can find a much more direct path to the global minimum rather than taking a much more convoluted path where you're sort of trying to follow a much more complicated trajectory to get to the global minimum.

### Feature Scaling

Idea: Make sure features are on a similar scale.

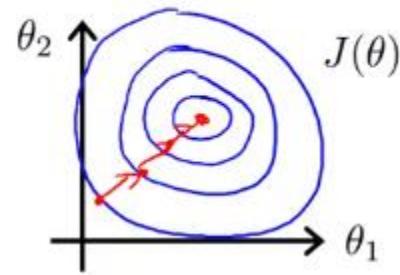
E.g.  $x_1 = \text{size (0-2000 feet}^2)$  ↪

$x_2 = \text{number of bedrooms (1-5)}$  ↪



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

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



So, **by scaling the features** so that there are **similar ranges of values**, in this example, we end up with both features,  $x_1$  and  $x_2$ , between zero and one. You can wind up with an implementation of **gradient descent**. They can **converge much faster**.

More generally, when we're performing **feature scaling**, what we often want to do is get **every feature** into approximately a **-1 to +1 range** and concretely, your feature  $x_0$  is always equal to 1. So, that's already in that range, but you may end up dividing other features by different numbers to get them to this range.

The numbers -1 and +1 aren't too important. So, if you have a feature,  $x_1$  that winds up being between zero and three, that's not a problem. If you end up having a different feature that winds being between -2 and +0.5, again, this is close enough to minus one and plus one that, you know, that's fine, and that's fine. It's only if you have a different feature, say  $x_3$  that is between, that ranges from -100 to +100, then, this is a very different range of values than minus 1 and plus 1. So, this might be a less well-scaled feature. And similarly, if your features take on a very, very small range of values so if  $x_4$  takes on values between minus 0.0001 and positive 0.0001, then again this takes on a much smaller range of values than the minus one to plus one range. And again I would consider this feature poorly scaled. So you want the range of values, you know, can be bigger than plus or smaller than plus one, but just not much bigger, like plus 100 here, or too much smaller like 0.001 over there.

Different people have different rules of thumb. But the one that I use is that if a feature takes on the range of values from say minus 3 to plus 3, I would think that should be just fine, but maybe it takes on much larger values than plus 3 or minus 3 then I start to worry and if it takes on values from say minus one-third to one-third. You know, I think that's fine too or 0 to one-third or minus one-third to 0. I guess that's typical range of values, something that's okay. But if it takes on a much tinier range of values like  $x_4$  here then again I start to worry.

So, the take-home message is don't worry if your features are not exactly on the same scale or exactly in the same range of values. But so long as they're all close enough to this gradient descent it should work okay.

In addition to dividing by so that the maximum value when performing feature scaling sometimes people will also do what's called mean normalization.

And what I mean by that is that you want to take a feature  $x_i$  and replace it with  $x_i - \mu_i$  to make your features have approximately 0 mean. And obviously we want to apply this to the feature  $x_0$ , because the feature  $x_0$  is always equal to one, so it cannot have an average value of zero. But concretely for other features if the range of sizes of the house takes on values between 0 to 2000 and if you know, the average size of a house is equal to 1000 then you might use this formula. Size, set the feature  $x_1$  to the size minus the average value divided by 2000 and similarly, on average if your houses have one to five bedrooms and if on average a house has two bedrooms then you might use this formula to mean normalize your second feature  $x_2$ . In both of these cases, you therefore wind up with features  $x_1$  and  $x_2$ . They can take on values roughly between minus .5 and positive .5. Exactly not true -  $x_2$  can actually be slightly larger than .5 but, close enough. And the more general rule is that you might take a feature  $x_1$  and replace it with  $x_1 - \mu_1 / S_1$ ; where to define these terms  $\mu_1$  is the average value of  $x_1$  in the training sets and  $S_1$  is the range of values of that feature, and by range, I mean let's say the maximum value minus the minimum value or for those of you that understand the deviation of the variable is setting  $S_1$  to be the standard deviation of the variable would be fine, too. But taking, you know, this max minus min would be fine. And similarly for the second feature,  $x_2$ , you replace  $x_2$  with this sort of subtract the mean of the feature and divide it by the range of values meaning the max minus min. And this sort of formula will get your features, you know, maybe not

exactly, but maybe roughly into these sorts of ranges, and by the way, for those of you that are being super careful technically if we're taking the range as max minus min this five here will actually become a four. So if max is 5 minus 1 then the range of their own values is actually equal to 4, but all of these are approximate and any value that gets the features into anything close to these sorts of ranges will do fine.

And the **feature scaling doesn't have to be too exact, in order to get gradient descent to run quite a lot faster.**

So, now you know about **feature scaling** and if you apply this simple trick, **it and make gradient descent run much faster** and **converge in a lot fewer iterations.**

That was feature scaling. In the next video, I'll tell you about another trick to make gradient descent work well in practice.

## Gradient Descent in Practice II - Learning Rate

In this video, I want to give you more practical tips for getting gradient descent to work.

The ideas in this video will center around the **learning rate, alpha.**

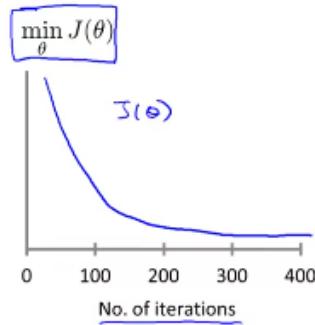
Concretely, here's the gradient descent update rule. And what I want to do in this video is **tell you about what I think of this debugging**, and some **tips for making sure that gradient descent is working correctly.** And **second**, I wanna tell you **how to choose the learning rate alpha** or at least how I go about choosing it.

### Gradient descent

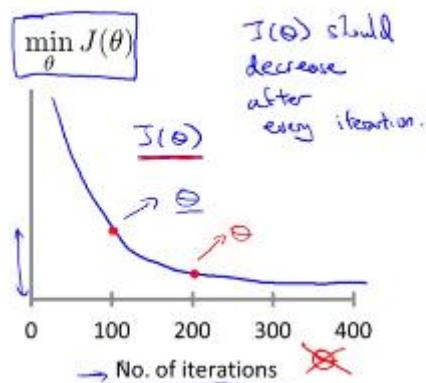
$$\rightarrow \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

- **“Debugging”:** How to make sure gradient descent is working correctly.
- How to choose learning rate  $\alpha$ .

Here's something that I often do to make sure that gradient descent is working correctly. **The job of gradient descent is to find the value of  $\theta$  for you that hopefully minimizes the cost function  $J(\theta)$ .** What I often do is therefore plot the cost function  $J(\theta)$  as gradient descent runs. So the x axis here is a number of iterations of gradient descent and as gradient descent runs you hopefully get a plot that maybe looks like this.

**Making sure gradient descent is working correctly.**

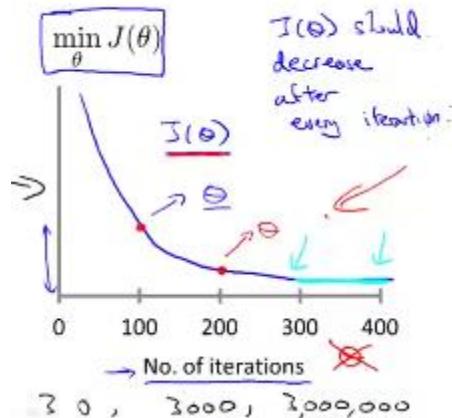
Notice that the x axis is number of iterations. Previously we were looking at plots of  $J(\theta)$  where the x axis, where the horizontal axis, was the **parameter vector  $\theta$**  but this is not what this is. Concretely, what this point is, is I'm going to run gradient descent for 100 iterations.

**Making sure gradient descent is working correctly.**

And **whatever value I get for theta after 100 iterations**, I'm going to get some value of theta after 100 iterations, and **I'm going to evaluate the cost function  $J(\theta)$** . For the value of theta I get after 100 iterations, and this **vertical height is the value of  $J(\theta)$ , for the value of theta I got after 100 iterations of gradient descent**. And this point here that corresponds to the value of  $J(\theta)$  for the theta that I get after I've run gradient descent for 200 iterations. So **what this plot is showing is, it's showing the value of your cost function after each iteration of gradient decent**. And **if gradient is working properly then  $J(\theta)$  should decrease after every iteration**.

And one useful thing that this sort of plot can tell you also is that if you look at the specific figure that I've drawn, it looks like by the time you've gotten out to maybe 300 iterations, between 300 and 400 iterations, in this segment it looks like  $J(\theta)$  hasn't gone down much more. So by the time you get to 400 iterations, it looks like this curve has flattened out here. And so way out here 400 iterations, it looks like gradient descent has more or less converged because your cost function isn't going down much more.

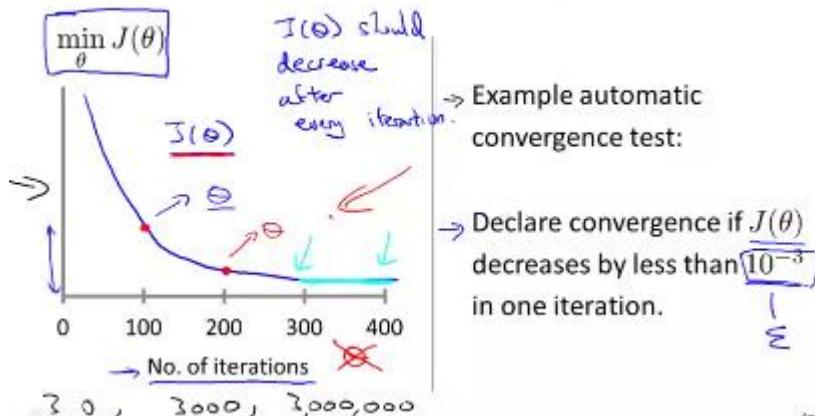
### Making sure gradient descent is working correctly.



So looking at this figure can also help you judge whether or not gradient descent has converged. By the way, the number of iterations the gradient descent takes to converge for a physical application can vary a lot, so maybe for one application, gradient descent may converge after just thirty iterations. For a different application, gradient descent may take 3,000 iterations, for another learning algorithm, it may take 3 million iterations. It turns out to be very difficult to tell in advance how many iterations gradient descent needs to converge.

And is usually by plotting this sort of plot, plotting the cost function as we increase in number in iterations, is usually by looking at these plots. But I try to tell if gradient descent has converged. It's also possible to come up with **automatic convergence test**, namely to have an algorithm that will tell you if gradient descent has converged. And here's maybe a pretty typical example of an automatic convergence test. And such a test may declare convergence if your cost function  $J(\theta)$  decreases by less than some small value epsilon, some small value  $10^{-3}$  in one iteration. But I find that usually choosing what this threshold is pretty difficult. And so in order to check your gradient descent's converge I actually tend to look at plots like these, like this figure on the left, rather than rely on an automatic convergence test. Looking at this sort of figure can also tell you, or give you an advance warning, if maybe gradient descent is not working correctly.

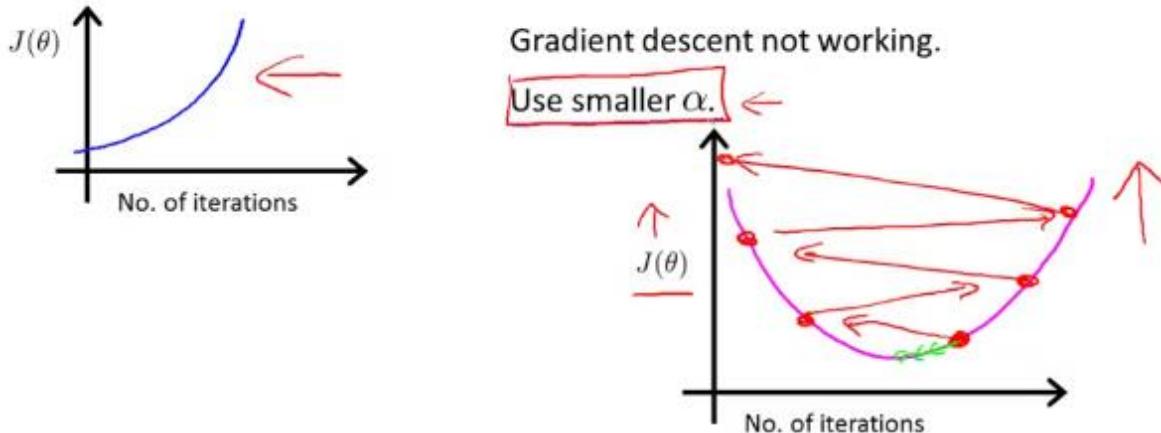
### Making sure gradient descent is working correctly.



Concretely, if you plot  $J(\theta)$  as a function of the number of iterations. Then if you see a figure like this where  **$J(\theta)$  is actually increasing**, then that gives you a clear sign that gradient descent is not working. And a theta like this usually means that **you should be using learning rate alpha**.

If  $J(\theta)$  is actually increasing, the **most common cause** for that is if you're maybe trying to minimize a function, that maybe looks like this. But if your **learning rate is too big** then if you start off there, **gradient descent may overshoot the minimum** and send you there. And if the learning rate is too big, you may overshoot again and it sends you there, and so on.

### Making sure gradient descent is working correctly.



So that, what you really wanted was for it to start here and for it to slowly go downhill, right? But **if the learning rate is too big, then gradient descent can instead keep on overshooting the minimum**. So that you actually end up getting worse and worse instead of getting to higher values of the cost function  $J(\theta)$ . So you end up with a plot like this and **if you see a plot like this, the fix is usually just to use a smaller value of alpha**. Oh, and also, of course, make sure your code doesn't have a bug in it. But usually **too large a value of alpha could be a common problem**.

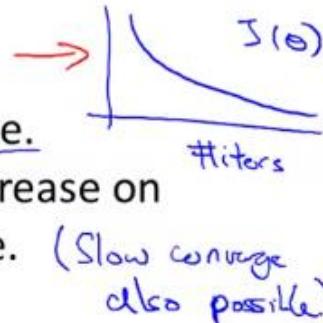
Similarly sometimes you may also see  $J(\theta)$  do something like this, it may go down for a while then go up, then go down for a while then go up, go down for a while go up and so on. And a **fix for something like this is also to use a smaller value of alpha**. I'm not going to prove it here, but under norm assumptions about the cost function  $J$ , that does hold true for linear regression, you can show that 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 set it smaller.

But of course, you also **don't want your learning rate to be too small** because if you do that **then the gradient descent can be slow to converge**. 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**.

**Summary:**

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



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 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.**

To choose  $\alpha$ , try

$$\dots, \underbrace{0.001}_{\uparrow}, \underbrace{0.003}_{\approx 3x}, \underbrace{0.01}_{\uparrow}, \underbrace{0.03}_{\approx 3x}, \underbrace{0.1}_{\uparrow}, \underbrace{0.3}_{\approx 3x}, \underbrace{1}_{\uparrow}, \dots$$

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 the value of alpha that seems to be causing  $J(\theta)$  to decrease rapidly. In fact, what I do actually isn't these steps of ten. So this is a scale factor of ten of each step up. 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 make sure that 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 reasonable value that I found. And when I do that usually it just gives me a good learning rate for my problem. And if you do this too, hopefully **you'll be able to choose a good learning rate** for your implementation of gradient descent.

## Features and Polynomial Regression

You now know about linear regression with multiple variables.

In this video, I wanna tell you a bit about the choice of features that you have and how you can get different learning algorithm, sometimes very powerful ones by choosing appropriate features. And in particular I also want to tell you about polynomial regression, which allows you to use the machinery of linear regression to fit very complicated, even very non-linear functions.

Let's take the example of predicting the price of the house. Suppose **you have two features**, the **frontage** of house and the **depth** of the house.

## Housing prices prediction

$$h_{\theta}(x) = \theta_0 + \underbrace{\theta_1 \times \text{frontage}}_{x_1} + \underbrace{\theta_2 \times \text{depth}}_{x_2}$$



So, here's the picture of the house we're trying to sell. So, the frontage is defined as this distance, is basically the width or the length of how wide your plot is if this plot of land that you own, and the depth of the house is how deep your property is, so there's a frontage, there's a depth, so you have two features called frontage and depth. You might build a linear regression model like this where frontage is your first feature  $x_1$  and depth is your second feature  $x_2$ , but when you're applying linear regression, you don't necessarily have to use just the features  $x_1$  and  $x_2$  that you're given.

What you can do is actually create new features by yourself. So, if I want to predict the price of a house, what I might do instead is decide that what really determines the size of the house is the **area** or the land area that I own. So, I might **create a new feature**. I'm just gonna call this feature  $x$  which is frontage, times depth.

Area

$$x = \underline{\text{frontage} * \text{depth}}$$

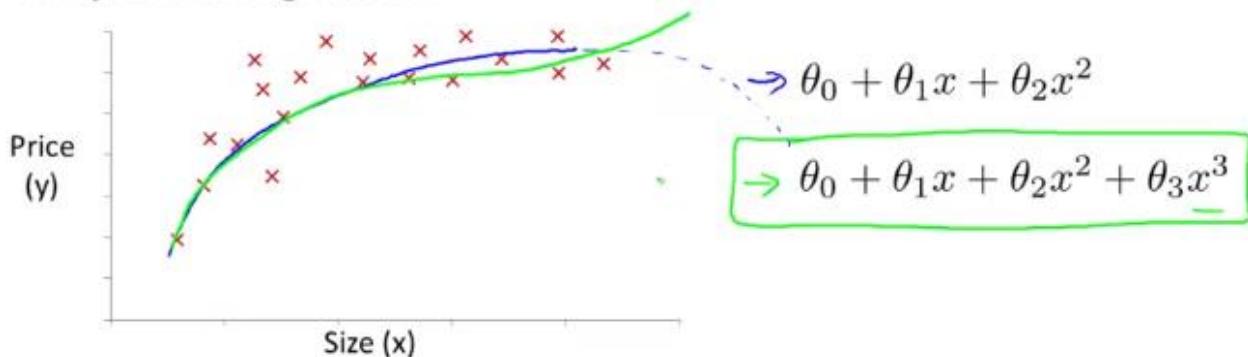
$$h_{\theta}(x) = \theta_0 + \theta_1 x$$

land area

This is a multiplication symbol. It's a frontage time's depth because this is the land area that I own and I might then select my hypothesis as that using just one feature which is my land area, right? Because the area of a rectangle is you know, the product of the length of the size. So depending on what insight you might have into a particular problem, rather than just taking the features frontage and depth that we happen to have started off with, **sometimes by defining new features you might actually get a better model**.

Closely related to the idea of choosing your features is this **idea called polynomial regression**.

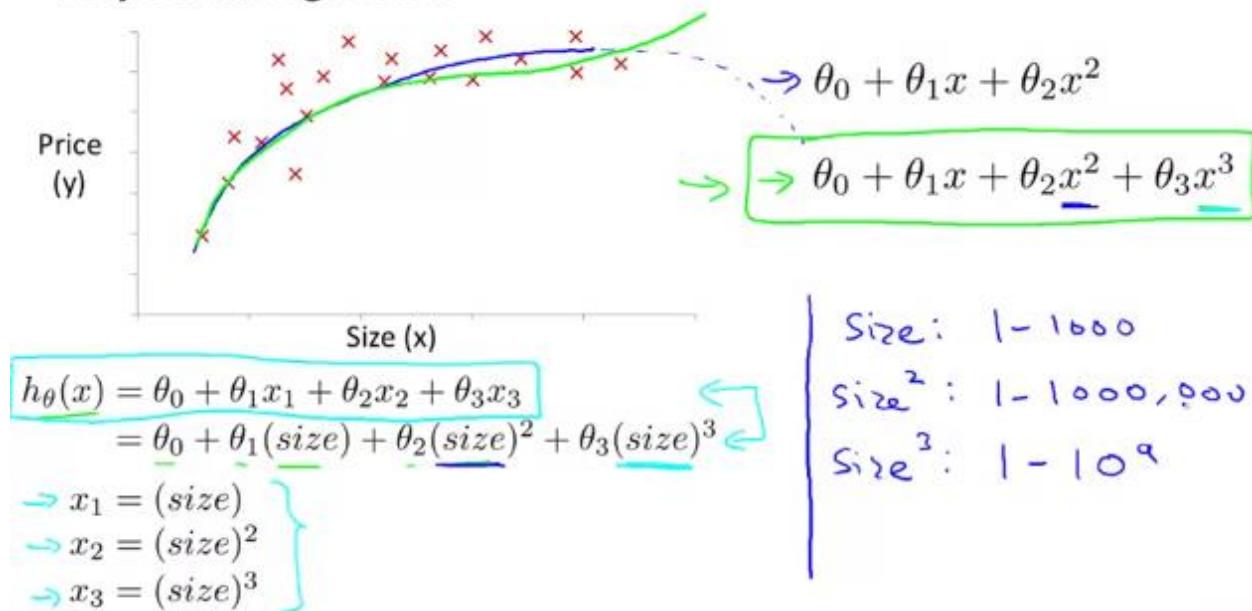
## Polynomial regression



Let's say you have a housing price data set that looks like this. Then there are a few different models you might fit to this. One thing you could do is fit a quadratic model like this. It doesn't look like a straight line fits this data very well. So maybe you want to fit a quadratic model like this where you think the size, where you think the price is a quadratic function and maybe that'll give you, you know, a fit to the data that looks like that.

But then you may decide that your quadratic model **doesn't make sense** because a quadratic function eventually, this function, comes back down and well, we don't think housing prices **should go down when the size goes up too high**. So then maybe we might choose a different polynomial model and choose to use instead a cubic function, and where we have now a third-order term and we fit that, maybe we get this sort of model, and maybe the green line is a somewhat better fit to the data because it doesn't eventually come back down. So how do we actually fit a model like this to our data?

## Polynomial regression



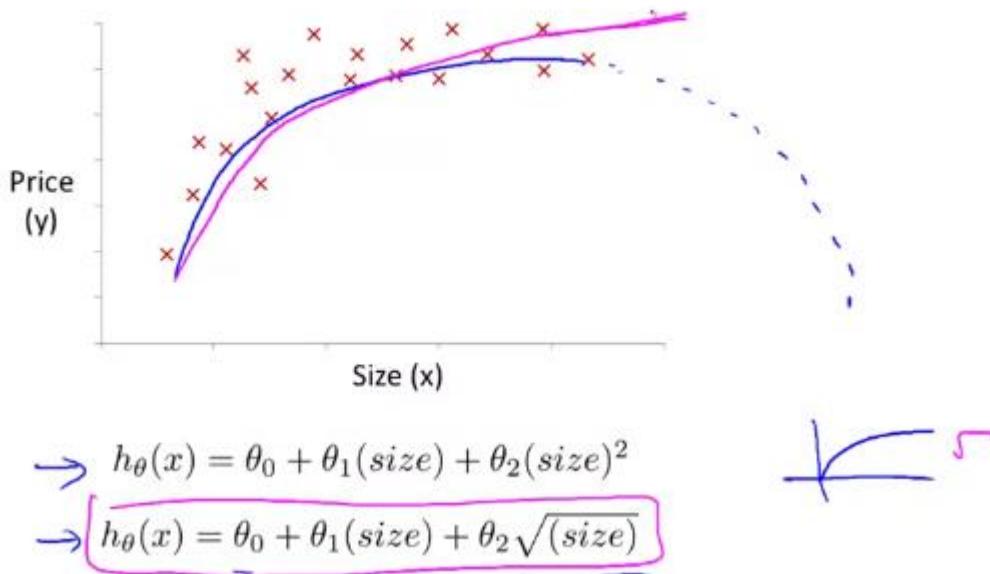
Using the machinery of **multivariate linear regression**, we can do this with a pretty simple modification to our algorithm. The form of the hypothesis we, we know how the fit looks like

this, where we say  $h(x)$  is  $\theta_0$  plus  $\theta_1x_1$  plus  $\theta_2x_2$  plus  $\theta_3x_3$ . And if we want to fit this cubic model that I have boxed in green, what we're saying is that to predict the price of a house, it's  $\theta_0$  plus  $\theta_1$  times the size of the house plus  $\theta_2$  times the square size of the house. So this term is equal to that term. And then plus  $\theta_3$  times the cube of the size of the house, raises that third term. In order to map these two definitions to each other, well, the natural way to do that is to set the first feature  $x_1$  to be the size of the house, and set the second feature  $x_2$  to be the square of the size of the house, and set the third feature  $x_3$  to be the cube of the size of the house. And, just by choosing my three features this way and applying the machinery of linear regression, I can fit this model and end up with a cubic fit to my data.

I just want to point out one more thing, which is that **if you choose your features like this, then feature scaling becomes increasingly important**. So if the size of the house ranges from one to a thousand, so, you know, from one to a thousand square feet, say, then the size squared of the house will range from one to one million, the square of a thousand, and your third feature  $x$  cubed, excuse me you, your third feature  $x_3$ , which is the size cubed of the house, will range from one two ten to the nine, and so these three features take on very different ranges of values, and **it's important to apply feature scaling if you're using gradient descent to get them into comparable ranges of values**.

Finally, here's one last example of how you really have broad choices in the features you use. 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 increases**. But rather than going to a cubic model there, you have, maybe, other choices of features and there are many possible choices.

## Choice of features



But just to give you another example of a reasonable choice, another reasonable choice might be to say that the price of a house is  $\theta_0$  plus  $\theta_1$  times the size, and then plus  $\theta_2$  times the square root

of the size, right? So the square root function is this sort of function, and maybe there will be some value of theta one, theta two, theta three, that will let you take this model and, for the curve that looks like that, and, you know, goes up, but sort of flattens out a bit and doesn't ever come back down. And, so, by having insight into, in this case, the shape of a square root function, and, into the shape of the data, by choosing different features, you can sometimes get better models.

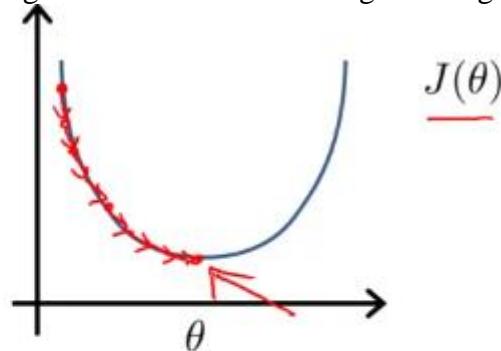
In this video, we talked about **polynomial regression**. That is, how to fit a polynomial, like a quadratic function, or a cubic function, to your data. Was also out this idea, that you have a choice in what features to use, such as that instead of using the frontage and the depth of the house, maybe, you can multiply them together to get a feature that captures the land area of a house. In case this seems a little bit bewildering, that with all these different feature choices, so how do I decide what features to use. **Later in this class, we'll talk about some algorithms where automatically choosing what features are used**, so you can have an algorithm look at the data and automatically choose for you whether you want to fit a quadratic function, or a cubic function, or something else. But, until we get to those algorithms for now I just want you to be aware that you have a choice in what features to use, and by designing different features you can fit more complex functions to your data than just fitting a straight line to the data and in particular you can put polynomial functions as well and sometimes by appropriate insight into the features simply get a much better model for your data.

## Normal Equation

In this video, we'll talk about the **normal equation**, which **for some linear regression problems**, will give us a much better way to solve for the optimal value of the parameters theta.

Concretely, so far the algorithm that we've been using for linear regression is gradient descent where **in order to minimize the cost function J of Theta**, we would take this iterative algorithm that takes many steps, multiple iterations of gradient descent to converge to the global minimum.

**Gradient Descent**



**Normal equation: Method to solve for  $\theta$  analytically.**

In contrast, the **normal equation would give us a method to solve for theta analytically**, so that rather than needing to run this iterative algorithm, we can instead just solve for the optimal value for theta all at one go, so that in basically one step you get to the optimal value right there.

It turns out the normal equation that has some advantages and some disadvantages, but before we get to that and talk about when you should use it, let's get some intuition about what this method does.

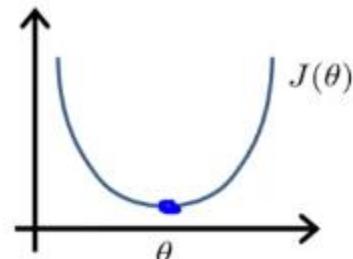
For this explanatory example, let's imagine, let's take a very simplified cost function  $J$  of Theta, that's just the function of a real number Theta. So, for now, imagine that Theta is just a scalar value or that Theta is just a raw value. It's just a number, rather than a vector. Imagine that we have a cost function  $J$  that's a quadratic function of this real valued parameter Theta, so  $J$  of Theta looks like that.

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

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

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

Solve for  $\theta$



Well, how do you minimize a quadratic function? For those of you that know a little bit of calculus, you may know that the way to minimize a function is to take derivatives and to set derivatives equal to zero. So, you take the derivative of  $J$  with respect to the parameter of Theta, you get some formula which I am not going to derive, and then you set that derivative equal to zero, and this allows you to solve for the value of Theta that minimizes  $J$  of Theta. That was a simpler case of when data was just real number.

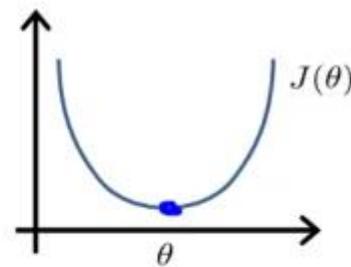
In the problem that we are interested in,  $\theta$  is no longer just a real number, but, instead, is this  $n+1$  dimensional parameter vector, and, a cost function  $J$  is a function of this vector value or  $\theta_0$  through  $\theta_n$ . And, a cost function looks like this, some square cost function on the right.

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

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

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

Solve for  $\theta$



$$\theta \in \mathbb{R}^{n+1}$$

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

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

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

How do we minimize this cost function  $J$ ? Calculus actually tells us that, if you, that one way to do so, is to take the partial derivative of  $J$ , with respect to every parameter of Theta  $J$  in turn, and

then, to set all of these to 0. If you do that, and you solve for the values of Theta 0, Theta 1, up to Theta N, then, this would give you the values of Theta to minimize the cost function J. Where, if you actually work through the calculus and work through the solution to the parameters Theta 0 through Theta N, the derivation ends up being somewhat involved. And, what I am going to do in this video, is actually to not go through the derivation, which is kind of long and kind of involved, but what I want to do is just tell you what you need to know in order to implement this process so you can solve for the values of the thetas that corresponds to where the partial derivatives is equal to zero. Or alternatively, or equivalently, the values of Theta is that minimize the cost function J of Theta. I realize that some of the comments I made that made more sense only to those of you that are normally familiar with calculus. So, but if you don't know, if you're less familiar with calculus, don't worry about it. I'm just going to tell you what you need to know in order to implement this algorithm and get it to work.

For the example that I want to use as a running example let's say that I have  $m = 4$  training examples. In order to implement this normal equation, what I'm going to do is the following.

### Examples

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

I'm going to take my data set, so here are my four training examples. In this case let's assume that, you know, these four examples is all the data I have. What I am going to do is take my data set and add an extra column that corresponds to my extra feature,  $x_0$ , that is always takes on this value of 1.

Examples:  $m = 4$ .

The diagram illustrates the preparation of data for a machine learning model. It shows a table of four training examples with columns for Size (feet<sup>2</sup>), Number of bedrooms, Number of floors, Age of home (years), and Price (\$1000). A green bracket underlines the first three columns, and a red bracket underlines the last column. A red arrow points to the first row, labeled  $x_0$ . Below the table, the data is represented as matrices  $X$  and  $y$ . Matrix  $X$  is defined as:

$$X = \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{bmatrix}$$

Below  $X$ , it is noted that it is  $m \times (n+1)$ . To the right of  $X$ , matrix  $y$  is shown as:

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

Below  $y$ , it is noted that it is an  $m$ -dimensional vector. At the bottom, the normal equation is given as:

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

What I'm going to do is I'm then going to construct a matrix called X that's a matrix that contains all of the features from my training data, so completely here is my, here are all my features and we're going to take all those numbers and put them into this matrix "X", okay? So just, you know, copy the data over one column at a time and then I am going to do something similar for y's. I am going to take the values that I'm trying to predict and construct now a vector, like so and call that a vector y. So X is going to be a m by (n+1) dimensional matrix, and y is going to be a m-dimensional vector where m is the number of training examples and n is, n is a number of features, n+1, because of this extra feature  $X_0$  that I had. Finally if you take your matrix X and you take your vector y, and if you just compute this, and set theta to be equal to **X transpose X inverse times X transpose y**, this would give you the value of theta that minimizes your cost function.

There was a lot that happened on the slides and I work through it using one specific example of one dataset. Let me just write this out in a slightly more general form and then let me just, and later on in this video let me explain this equation a little bit more. In case it is not yet entirely clear how to do this.

In a general case, let us say we have m training examples so  $x^1, y^1$  up to  $x^m, y^m$  and n features. So, each of the training example  $\underline{x}^{(i)}$  may look like a vector like this, that is an **n+1 dimensional feature vector**. The way I'm going to construct the matrix "X", this is also called the design matrix, is as follows.

**Each training example gives me a feature vector** like this, say sort of n+1 dimensional vector.

$$\begin{array}{c} \text{m examples } (\underline{x}^{(1)}, y^{(1)}), \dots, (\underline{x}^{(m)}, y^{(m)}) ; n \text{ features.} \\ \hline \underline{x}^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix} \in \mathbb{R}^{n+1} \quad | \quad \begin{array}{l} X = \begin{bmatrix} \cdots & (\underline{x}^{(1)})^\top & \cdots \\ \cdots & (\underline{x}^{(2)})^\top & \cdots \\ \vdots & \vdots & \vdots \\ \cdots & (\underline{x}^{(m)})^\top & \cdots \end{bmatrix} \\ (\text{design matrix}) \end{array} \\ \text{E.g. If } \underline{x}^{(i)} = \begin{bmatrix} 1 \\ x_1^{(i)} \end{bmatrix} \quad \underline{x} = \begin{bmatrix} 1 & \underline{x}_1^{(1)} \\ 1 & \underline{x}_2^{(1)} \\ \vdots & \vdots \\ 1 & \underline{x}_m^{(1)} \end{bmatrix} \quad \underline{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \quad \text{is } \underline{\underline{\theta}} = (\underline{x}^\top \underline{x})^{-1} \underline{x}^\top \underline{y} \end{array}$$

**The way I am going to construct my design matrix X** is only construct the matrix like this, and what I'm going to do is take the first training example, so that's a vector, take its transpose so it ends up being this, you know, long flat thing and make  $x^1$  transpose the first row of my design matrix. Then I am going to take my second training example,  $x^2$ , take the transpose of that and

put that as the second row of  $X$  and so on, down until my last training example. Take the transpose of that, and that's my last row of my matrix  $X$ . And, so, **this makes my matrix  $X$ , an  $m$  by  $n+1$  dimensional matrix**. As a concrete example, let's say I have only one feature, really, only one feature other than  $x^0$ , which is always equal to 1. So if my feature vectors  $x^{(i)}$  are equal to this 1, which is  $x^0$ , then some real feature, like maybe the size of the house, then my design matrix,  $X$ , would be equal to this.

For the first row, I'm going to basically take this and take its transpose. So, I'm going to end up with 1, and then  $x^{(1)}_1$ . For the second row, we're going to end up with 1 and then  $x^{(1)}_2$  and so on down to 1, and then  $x^{(1)}_m$ . And thus, this will be an  $m \times 2$  dimensional matrix. So, that's how to construct the matrix  $X$ . And, the vector  $y$  may be at sometimes I might write an arrow on top to denote that it is a vector, but very often I'll just write this as  $y$ , either way. The vector  $y$  is obtained by taking all the labels, all the correct prices of houses in my training set, and just stacking them up into an  $m$ -dimensional vector, and that's  $y$ .

Finally, having constructed the matrix  $X$  and the vector  $y$ , we then just **compute theta as  $(X$  transpose  $X$ ) inverse times  $X$  transpose  $y$** .

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

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

Set  $A = \underline{X^T X}$

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

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

I just want to make sure that this equation makes sense to you and that you know how to implement it. So, you know, concretely, what is this  $(X^T X)^{-1}$ ? Well,  $(X^T X)^{-1}$  is the inverse of the matrix  $X^T X$ .

Concretely, if you were to say set  $A$  to be equal to  $X^T X$ , so  $X'$  is a matrix,  $X^T X$  gives you another matrix, and we call that matrix  $A$ . Then, you know,  $(X^T X)^{-1}$  is just you **take this matrix  $A$  and you invert it**, right! This gives, let's say  $A^{-1}$ . And so that's how you compute this thing. You compute  $X^T X$  and then you compute its inverse.

We haven't yet talked about Octave. We'll do so in the later set of videos, but in the Octave programming language or a similar software, and also the MATLAB programming language is very similar. The command to compute this quantity,  $(X^T X)^{-1} X^T Y$  is as follows:

Octave:  $\text{pinv}(X' * X) * X' * y$   $X'$   $X^T$

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

$$\theta = \text{pinv}(X^T * X)^{-1} * X^T * y \quad \min J(\theta)$$

In Octave X prime is the notation that you use to denote X transpose. And so, this expression that's boxed in red, that's computing X transpose times X. pinv is a function for computing the inverse of a matrix, so this computes X transpose X inverse, and then you multiply that by X transpose, and you multiply that by Y. So you end computing that formula which I didn't prove, but it is possible to show mathematically even though I'm not going to do so here, that this formula gives you the optimal value of theta in the sense that if you set theta equal to this, that's the value of theta that minimizes the cost function J of theta for the new regression.

One last detail, in the earlier video I talked about the **feature scaling** and the idea of getting features to be on similar ranges of scales, of similar ranges of values of each other.

~~Feature Scaling~~

$$0 \leq x_1 \leq 1$$

$$0 \leq x_2 \leq 1000$$

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

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 are some of their advantages and disadvantages.

Let's say you have **m training examples** and **n features**. One **disadvantage of gradient descent is that, you need to choose the learning rate Alpha**. And, often, this means running it few times with different learning rate alphas and then seeing what works best. And so that is sort of extra work and extra hassle. **Another disadvantage with gradient descent is it needs many more iterations**. So, depending on the details, that could make it slower, although there's more to the story as we'll see in a second. As for the normal equation, you don't need to choose any learning rate alpha. So that, you know, makes it really convenient, makes it simple to implement. You just run it and it usually just works. And **you don't need to iterate, so, you don't need to plot J of Theta or check the convergence or take all those extra steps**. So far, the balance seems to favor normal, the normal equation.

Here are some disadvantages of the normal equation, and some advantages of gradient descent.

**Gradient descent works pretty well, even when you have a very large number of features.**

So, even if you have millions of features you can run gradient descent and it will be reasonably efficient. It will do something reasonable. In contrast to normal equation in order to solve for the parameters theta, we need to solve for this term. We need to compute this term,  $X^T X$ ,  $X^{-1}$ . This matrix  $X^T X$ . That's an  $n \times n$  matrix, if you have  $n$  features. Because, if you look at the dimensions of  $X^T X$  the dimension of  $X$ , you multiply, figure out what the dimension of the product is, the matrix  $X^T X$  is an  $n \times n$  matrix where  $n$  is the number of features, and for almost computed implementations the cost of inverting the matrix, rose roughly as the cube of the dimension of the matrix. So, computing this inverse costs, roughly order  **$n$  cube time**. Sometimes, it's slightly faster than  $n$  cube but, it's, you know, close enough for our purposes. So **if  $n$  is the number of features is very large, then computing this quantity can be slow and the normal equation method can actually be much slower**. So if  $n$  is large then I might usually use gradient descent because we don't want to pay this all  $n$  cube time. 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 kind of slow, and I might then start to maybe lean in the direction of gradient descent, but maybe not quite.  $n$  equals ten thousand, you can sort of convert a ten-thousand-by-ten-thousand matrix. But if it gets much bigger than that, then, I would probably use gradient descent. **So, if  $n$  equals ten to the sixth with a million features, then inverting a million-by-million matrix is going to be very expensive, and I would definitely favor gradient descent** if you have that many features.

So exactly how large set of features has to be before you convert a gradient descent, it's hard to give a strict number. But, **for me, it is usually around ten thousand** that I might start to consider switching over to gradient descents or maybe, some other algorithms that we'll talk about later in this class. To summarize, so long as the number of features is not too large, the normal equation gives us a great alternative method to solve for the parameter theta.

Concretely, so long as the number of features is less than 1000, you know, I would use, I would usually use in normal equation method rather than, gradient descent. To preview some ideas that we'll talk about later in this course, as we get to the more complex learning algorithm, for example, when we talk about **classification algorithm**, like a **logistic regression** algorithm, We'll see that those algorithm actually... the normal equation method actually does not work for those more sophisticated learning algorithms, and, we will have to resort to gradient descent for those algorithms.

So, gradient descent is a very useful algorithm to know. Both for linear regression when you have a large number of features and for some of the other algorithms that we'll see in this course, because, for them, the normal equation method just doesn't apply and doesn't work. But for this specific model of linear regression, the normal equation can give you an alternative that can be much faster, than gradient descent. So, depending on the detail of your algorithm, depending on the detail of the problems and how many features that you have, both of these algorithms are well worth knowing about.

## Normal Equation Noninvertibility

In this video I want to talk about the **Normal equation and non-invertibility**.

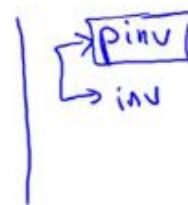
This is a somewhat more advanced concept, but it's something that I've often been asked about. And so I want to talk it here and address it here. But this is a somewhat more advanced concept, so feel free to consider this optional material. And there's a phenomenon that you may run into that may be somewhat useful to understand, but even if you don't understand the normal equation and linear progression, you should really get that to work okay. Here's the issue.

### Normal equation

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

- What if  $X^T X$  is non-invertible? (singular/degenerate)

- Octave:  $\text{pinv}(X' * X) * X' * y$



For those of you there are, maybe some are more familiar with linear algebra, what some students have asked me is, when computing this Theta equals X transpose X inverse X transpose Y. What if the matrix X transpose X is non-invertible?

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

- Redundant features (linearly dependent).

E.g.  $\begin{cases} x_1 = \text{size in feet}^2 \\ \cancel{x_2 = \text{size in m}^2} \\ \hline x_1 = (3.28)^2 x_2 \end{cases}$   $l_m = 3.28 \text{ feet}$

$$\begin{array}{l} \rightarrow m = 10 \leftarrow \\ \rightarrow n = 100 \leftarrow \\ \qquad \qquad \qquad \mathcal{O} \in \mathbb{R}^{101} \end{array}$$

- Too many features (e.g.  $m \leq n$ ).

- Delete some features, or use regularization.

$\downarrow$  later

So for those of you that know a bit more linear algebra you may know that only some matrices are invertible and some matrices do not have an inverse we call those non-invertible matrices Singular or Degenerate matrices. The issue or the problem of  $x^T x$  being non-invertible should happen pretty rarely.

And in Octave if you implement this to compute theta, it turns out that this will actually do the right thing. I'm getting a little technical now, and I don't want to go into the details, but Octave has two functions for inverting matrices. One is called `pinv`, and the other is called `inv`. And the differences between these two are somewhat technical. One's called the pseudo-inverse, one's called the inverse. But you can show mathematically that so long as you use the `pinv` function then this will actually compute the value of data that you want even if  $X^T X$  is non-invertible. The specific details between `inv`, what is the difference between `pinv`? What is `inv`? That's somewhat advanced numerical computing concepts, I don't really want to get into. But I thought in this optional video, I'll try to give you little bit of intuition about what it means for  $X^T X$  to be non-invertible. For those of you that know a bit more linear Algebra might be interested.

I'm not gonna prove this mathematically but if  $X^T X$  is non-invertible, there usually two most common causes for this. The first cause is if somehow in your learning problem you have redundant features. Concretely, if you're trying to predict housing prices and if  $x_1$  is the size of the house in feet, in square feet and  $x_2$  is the size of the house in square meters, then you know 1 meter is equal to 3.28 feet Rounded to two decimals. And so your two features will always satisfy the constraint  $x_1$  equals 3.28 squared times  $x_2$ . And you can show for those of you that are somewhat advanced in linear Algebra, but if you're explaining the algebra you can actually show that if your two features are related, are a linear equation like this then matrix  $X^T X$  would be non-invertible. The second thing that can cause  $X^T X$  to be non-invertible is if you are training, if you are trying to run the learning algorithm with a lot of features.

Concretely, if  $m$  is less than or equal to  $n$ . For example, if you imagine that you have  $m = 10$  training examples that you have  $n$  equals 100 features then you're trying to fit a parameter back to theta which is, you know,  $n$  plus one dimensional. So this is 101 dimensional, you're trying to fit 101 parameters from just 10 training examples. This turns out to sometimes work but not always be a good idea. Because as we'll see later, you might not have enough data if you only have 10 examples to fit you know, 100 or 101 parameters. We'll see later in this course why this might be too little data to fit this many parameters. But commonly what we do then if  $m$  is less than  $n$ , is to see if we can either delete some features or to use a technique called regularization which is something that we'll talk about later in this course as well, that will kind of let you fit a lot of parameters, use a lot features, even if you have a relatively small training set. But this regularization will be a later topic in this course.

But to summarize if ever you find that  $X^T X$  is singular or alternatively you find it non-invertible, what I would recommend you do is first look at your features and see if you have redundant features like this  $x_1, x_2$ , you're being linearly dependent or being a linear function of each other like so. And if you do have redundant features and if you just delete one of these features, you really don't need both of these features. If you just delete one of these

features that would solve your non-invertibility problem. And so I would first think through my features and check if any are redundant. And if so then **keep deleting redundant features until they're no longer redundant**. And if your features are not redundant, I would check if **I may have too many features**. And if that's the case, **I would either delete some features** if I can bear to use fewer features or **else I would consider using regularization**. Which is this topic that we'll talk about later.

So that's it for the normal equation and what it means for if the matrix  $X$  transpose  $X$  is non-invertible but this is a problem that you should run that hopefully you run into pretty rarely and if you just implement it in octave using **pinv**, and using the pinv function, which is called a pseudo inverse function, so you could use a different linear algebra library it is called a pseudo-inverse but that implementation should just do the right thing, even if  $X$  transpose  $X$  is non-invertible, which should happen pretty rarely anyways, so this should not be a problem for most implementations of linear regression.

## Octave Basic Operations

You now know a bunch about machine learning.

In this video, I like to teach you a **programming language, Octave**, in which you'll be able to very quickly **implement the learning algorithms** we've seen already, and the learning algorithms we'll see later in this course.

In the past, I've tried to teach machine learning using a large variety of different programming languages including C++ Java, Python, NumPy, and also Octave, and what I found was that students were able to learn the most productively learn the most quickly and prototype your algorithms most quickly using a relatively high level language like octave.

In fact, what I often see in Silicon Valley is that if even if you need to build. If you want to build a large scale deployment of a learning algorithm, what people will often do is prototype and the language is Octave. Which is a great prototyping language.

So you can sort of **get your learning algorithms working quickly**. And then only if you need to a very large scale deployment of it. Only then spend your time **re-implementing the algorithm to C++ Java or some of the language** like that.

Because all the lessons we've learned is that a time or develop a time. That is your time. The machine learning's time is incredibly valuable. And if you can get your learning algorithms to work more quickly in Octave. Then overall you have a huge time savings by first developing the algorithms in Octave, and then implementing and maybe C++ Java, only after we have the ideas working.

The most common prototyping language I see people use for machine learning are: **Octave, MATLAB, Python, NumPy, and R.**

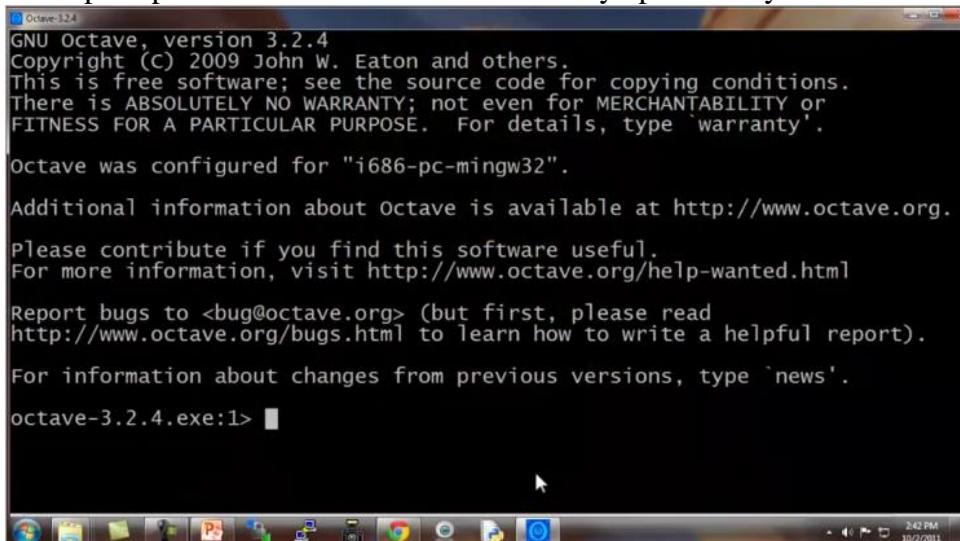
Octave is nice because open sourced. And MATLAB works well too, but it is expensive for too many people. But if you have access to a copy of MATLAB. You can also use MATLAB with this class. If you know **Python**, NumPy, or if you know R. I do see some people use it. But, what I see is that people usually end up developing somewhat more slowly, and you know, these languages. Because the **Python, NumPy syntax is just slightly clunkier (dictionary meaning – heavy) than the Octave syntax**. And so because of that, and because we are releasing starter code in Octave. **I strongly recommend that you not try to do the following exercises in this class in NumPy and R.** But that I do recommend that you instead do the programming exercises for this class in octave instead.

What I'm going to do in this video is go through a list of commands very, very quickly, and its goal is to quickly show you the range of commands and the range of things you can do in Octave. The course website will have a transcript of everything I do, and so after watching this video **you can refer to the transcript posted on the course website when you want find a command.** Concretely, what I recommend you do is first watch the tutorial videos.

### **And after watching to the end, then install Octave on your computer.**

And finally, it goes to the course website, download the transcripts of the things you see in the session, and type in whatever commands seem interesting to you into Octave, so that it's running on your own computer, so you can see it run for yourself. And with that let's get started.

Here's my Windows desktop, and I'm going to **start up Octave**. And I'm now in Octave. And that's my Octave prompt. Let me first show the elementary operations you can do in Octave.



```
Octave-3.2.4
GNU Octave, version 3.2.4
Copyright (C) 2009 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. For details, type `warranty'.
Octave was configured for "i686-pc-mingw32".
Additional information about Octave is available at http://www.octave.org.
Please contribute if you find this software useful.
For more information, visit http://www.octave.org/help-wanted.html
Report bugs to <bug@octave.org> (but first, please read
http://www.octave.org/bugs.html to learn how to write a helpful report).
For information about changes from previous versions, type `news'.
octave-3.2.4.exe:1> █
```

So you type in **5 + 6**. That gives you the answer of **11**.

```
octave-3.2.4.exe:1> 5+6
ans = 11
octave-3.2.4.exe:2> █
```

**3 - 2**

```
octave-3.2.4.exe:2> 3-2
ans = 1
```

```
5 x 8
octave-3.2.4.exe:3> 5*8
ans = 40
```

```
1/2
octave-3.2.4.exe:4> 1/2
ans = 0.50000
```

$2^6$  is 64.

```
octave-3.2.4.exe:5> 2^6
ans = 64
octave-3.2.4.exe:6>
```

So those are the elementary math operations.

You can also do **logical operations**.

So one equals two. This evaluates to false.

The percent command here means a comment.

```
octave-3.2.4.exe:6>
octave-3.2.4.exe:6>
octave-3.2.4.exe:6> 1 == 2 % false
ans = 0
```

So, one equals two, evaluates to false. Which is represented by zero. One not equals to two. This is true. So that returns one.

Note that a **not equal sign** is this tilde equals symbol.

```
octave-3.2.4.exe:7> 1 ~= 2
ans = 1
```

And not bang **equals**. Which is what some other programming languages use.

Lets see **logical operations** one and zero use a double ampersand sign to the **logical AND**. And that evaluates false.

```
octave-3.2.4.exe:8> 1 && 0 % AND
ans = 0
```

One or zero is the **OR** operation. And that evaluates to true.

```
octave-3.2.4.exe:9> 1 || 0 % OR
ans = 1
```

And I can **XOR** one and zero, and that evaluates to one.

```
octave-3.2.4.exe:10> xor(1,0)
ans = 1
```

This thing over on the left, this **Octave 324.exe:11>**, this is the **default Octave prompt**.

```
octave-3.2.4.exe:11>
```

It shows the, what, the version in Octave and so on. If you don't want that prompt, there's a somewhat cryptic command **PS1 ('>> ')**; and so on, that you can use to change the prompt.

```
octave-3.2.4.exe:11> PS1('>> ');
>>
>>
>>
```

That's what I prefer my Octave prompt to look like. So if I hit enter.

**Now my Octave prompt has changed to the greater than, greater than sign.** Which, you know, looks quite a bit better.

## Variables

Next let's talk about **Octave variables**.

I can take the variable a and assign it to 3.

And hit enter.

And now a is equal to 3.

```
>> a = 3
a = 3
```

## Semicolon

You want to assign a variable, but you don't want to print out the result. If you put a semicolon, the **semicolon suppresses the print output**. So to do that, enter, it doesn't print anything.

```
>> a = 3; % semicolon supressing output
>>
```

Whereas A equals 3. Mix it, print it out, where A equals, 3 semicolon doesn't print anything.

## String

I can do **string assignment**.

B equals hi

Now if I just enter B it prints out the variable B. So B is the string hi

```
>> b = 'hi';
>> b
b = hi
>>
```

## Variable Assignment and Printing

C equals 3 greater than equal 1.

So, now C evaluates to true.

```
>> c = (3>=1);
>> c
c = 1
>>
```

**If you want to print out or display a variable**, here's how you go about it.

Let me set a equals Pi.

And if I want to print a I can just type a like so, and it will print it out.

```
>>
>> a=pi;
>> a
a = 3.1416
```

## Display Formatting

For more complex printing there is also the **disp** command which stands for Display. disp a just prints out a. You can also display strings.

**disp(sprintf('2 decimals: %0.2f', a)).**

And this will print out the string. Two decimals, colon, 3.14.

```
a = 3.1416
>> disp(a);
3.1416
>> disp(sprintf('2 decimals: %0.2f', a))
2 decimals: 3.14
>>
```

This is kind of an old style C syntax. For those of you that have programmed C before, this is essentially the syntax you use to print screen. So the sprintf generates a string that is less than the 2 decimals, 3.1 plus string. This percent 0.2 F means substitute A into here, showing the two digits after the decimal points.

And to show you another example,

**sprintf six decimals percent 0.6 F comma a.**

And, this should print Pi with six decimal places.

```
>> disp(sprintf('6 decimals: %0.6f', a))
6 decimals: 3.141593
>> a
a = 3.1416
>>
```

Finally, I was saying, a like so, looks like this. There are useful shortcuts that type **formats long**.

It causes strings by default be displayed to a lot more decimal places.

```
>> format long
>> a
a = 3.14159265358979
>>
```

And **format short** is a command that restores the default of just printing a small number of digits.

```
>> format short
>> a
a = 3.1416
>>
```

Okay, that's how you work with variables.

## Vectors and Matrices

Now let's look at **vectors** and **matrices**.

Let's say I want to assign **A** to the **matrix**.

Let me show you an example: 1, 2, semicolon, 3, 4, semicolon, 5, 6.

This generates a three by two matrix A whose first row is 1, 2. Second row 3, 4. Third row is 5, 6.

```
>>
>>
>> A = [1 2; 3 4; 5 6]
A =
1 2
3 4
5 6
>>
```

What the **semicolon** does is essentially say, go to the next row of the matrix.

There are other ways to type this in.

Type A 1, 2 semicolon 3, 4, semicolon, 5, 6, like so.

```
>> A = [1 2;
> 3 4;
> 5 6]
A =
1 2
3 4
5 6
>>
```

And that's another equivalent way of assigning A to be the values of this three by two matrix.

Similarly you can assign **vectors**.

So **V** equals 1, 2, 3.

This is actually a **row vector**.

```
>>
>> v = [1 2 3]
v =
1 2 3
>>
```

This is a 1 by 3 matrix, right. Not 3 by 1.

If I want to assign this to a **column vector**, what I would do instead is do v 1;2;3.

And this will give me a 3 by 1.

```
>> v = [1; 2; 3]
v =
1
2
3
>>
```

So this will be a column vector.

Here's some more useful notation.

**V equals 1:0.1:2.**

What this does is it sets V to the bunch of elements that start from 1.

And increments and steps of 0.1 until you get up to 2.

So if I do this, V is going to be this, you know, **row vector**.

This is what **one by eleven matrix** really.

That's 1, 1.1, 1.2, 1.3 and so on until we get up to two.

```
>>
>> v = 1:0.1:2
v =
Columns 1 through 7:
1.0000    1.1000    1.2000    1.3000    1.4000    1.5000    1.6000
Columns 8 through 11:
1.7000    1.8000    1.9000    2.0000
>>
```

Now, and I can also set **V equals one colon six**, and that sets V to be these numbers 1 through 6, okay.

```
>>
>> v = 1:6
v =
1   2   3   4   5   6
>>
```

Now here are **some other ways to generate matrices**.

**ones(2, 3)** is a command that generates a matrix that is a two by three matrix that is the matrix of all ones.

```
>>
>> ones(2,3)
ans =
1   1   1
1   1   1
>>
```

So if I set that **c = 2 \* ones(2, 3)** this generates a two by three matrix that is all two's.

You can think of this as a shorter way of writing this and c2,2,2's and you can call them 2,2,2, which would also give you the same result.

```
>> c = 2*ones(2,3)
c =
2   2   2
2   2   2
>>
```

Let's say **w = ones(1, 3)**, so this is going to be a **row vector** or a row of three one's.

```
>> w = ones(1, 3)
w =
    1   1   1
```

And similarly you can also say **w = zeros(1,3)**, and this generates a matrix. A one by three matrix of all zeros.

```
>> w = zeros(1, 3)
w =
    0   0   0
>>
```

Just a couple more ways to generate **matrices**.

If I do **w = rand(1, 3)**, this gives me a one by three matrix of all random numbers.

```
>> w = rand(1, 3)
w =
    0.91477   0.14359   0.84860
>>
```

If I do **rand three by three**. This gives me a three by three matrix of all random numbers drawn from the uniform distribution between zero and one.

```
>> rand(3, 3)
ans =
    0.390426   0.264057   0.683559
    0.041555   0.314703   0.506769
    0.521893   0.739979   0.387001
>>
```

So **every time I do this, I get a different set of random numbers** drawn uniformly between zero and one.

```

>> rand(3,3)
ans =
0.390426  0.264057  0.683559
0.041555  0.314703  0.506769
0.521893  0.739979  0.387001

>> rand(3,3)
ans =
0.467747  0.684916  0.346052
0.022935  0.603373  0.307135
0.212884  0.857236  0.456541

>> rand(3,3)
ans =
0.082306  0.450805  0.307135
0.218295  0.554723  0.819940
0.728084  0.893041  0.312381
>>

```

For those of you that know what a **Gaussian random** variable is or for those of you that know what a normal random variable is, you can also set **w = randn(1,3)**, one by three. And so these are going to be three values drawn from a Gaussian distribution with **mean zero** and **variance or standard deviation equal to one**.

```

>> w = randn(1,3)
w =
-0.33517   1.26847  -0.28211
>>

```

And you can set **more complex things** like **w = -6 + sqrt(10)\*randn(1, 10000);** And I'm going to put a **semicolon** at the end because I **don't really want this printed out**. This is going to be a what?

Well, it's going to be a vector of, with a hundred thousand, excuse me, ten thousand elements.

```

>>
>> w = -6 + sqrt(10)*(randn(1,10000));

```

So, well, actually, you know what? **Let's print it out**. So this will generate a matrix like this. Right? With 10,000 elements. So that's what W is.

```

>>
>> w = -6 + sqrt(10)*(randn(1,10000))
>>

```

```

W =
Columns 1 through 6:
-5.0865e+000 -6.2714e+000 -5.2154e+000 -1.9512e+000 -6.7351e+000 -1.
0561e+001

Columns 7 through 12:
-1.0258e+001 -4.6307e+000 -4.2545e+000 -2.8252e+000 -8.5130e+000 -2.
5872e+000

Columns 13 through 18:
-8.3948e+000 -7.8885e+000 -5.0174e+000 -8.6418e+000 -9.7672e+000 -6.
7775e+000

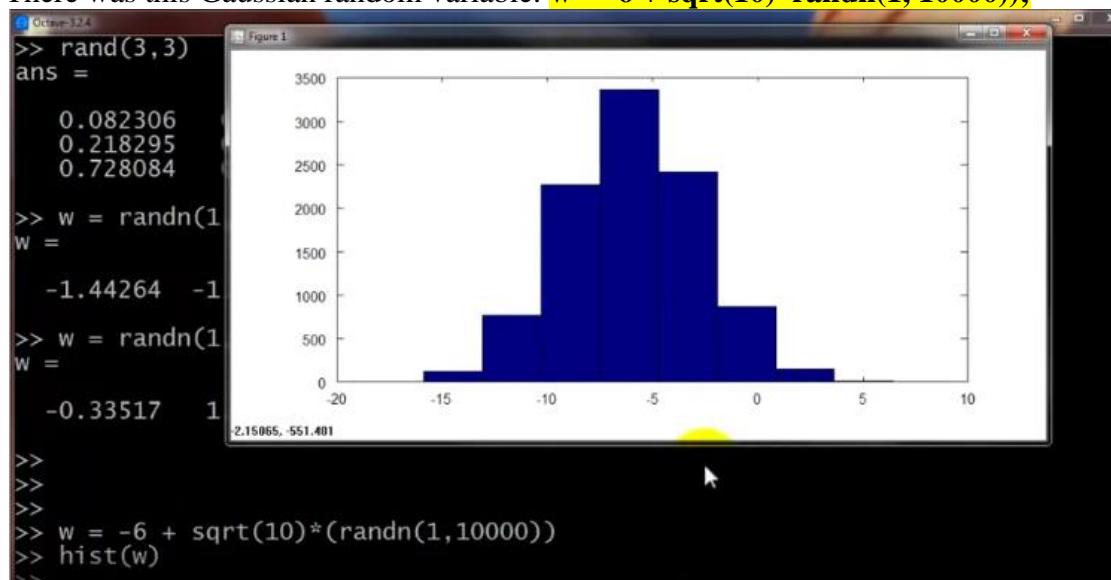
Columns 19 through 24:
-1.1129e+001 -3.1476e+000 -3.7968e+000 -7.4720e+000 -3.0153e+000 -7.
9433e+000

lines 1-18 -- (f)orward, (b)ack, (q)uit

```

## Histogram

And if I now plot a **histogram** of W with a **hist** command. And Octave's print hist command, you know, takes a couple seconds to bring this up, but this is a histogram of my random variable for W. There was this Gaussian random variable. **w = -6 + sqrt(10)\*randn(1, 10000);**



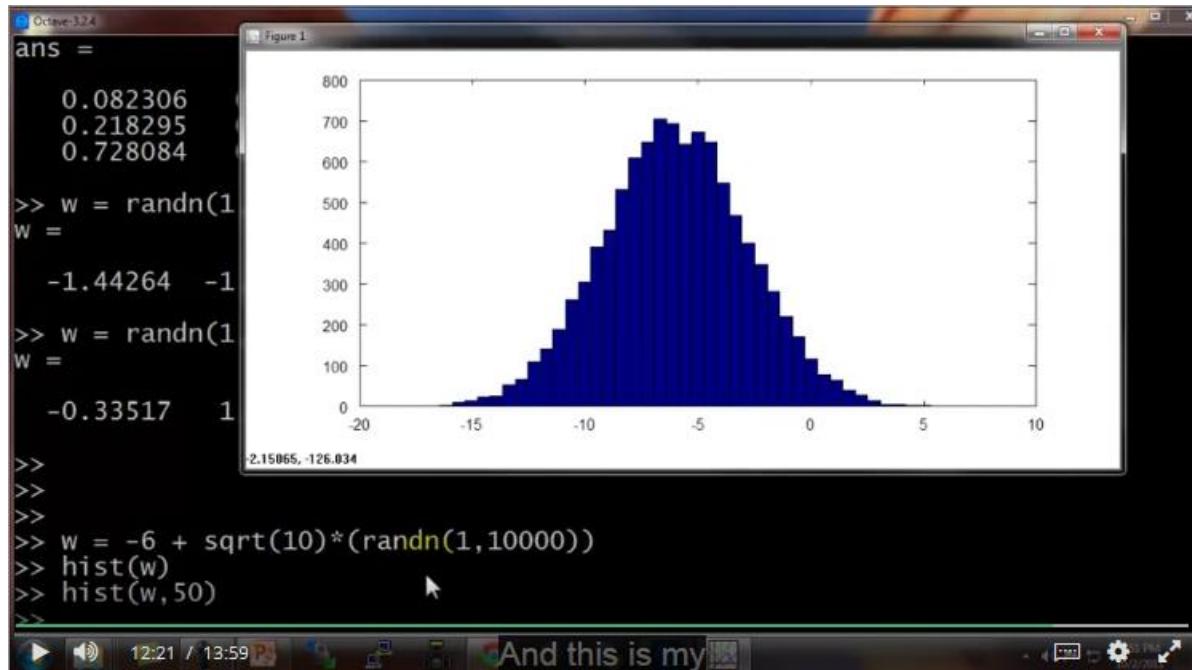
And I can plot a **histogram with more buckets, with more bins, with say, 50 bins**.

And this is my histogram of a Gaussian with mean minus 6.

Because I have a minus 6 there plus square root 10 times this.

So the variance of this Gaussian random variable is 10 on the standard deviation is square root of 10, which is about what?

Three point one.



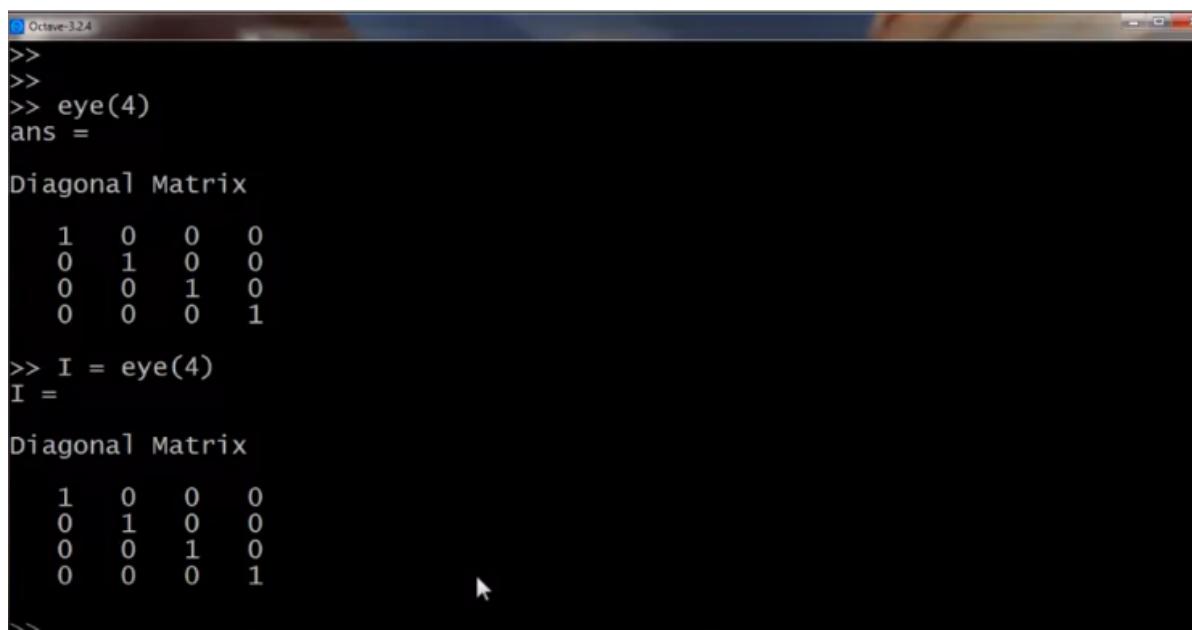
## eye() command

Finally, one special command for **generator matrix**, which is the **eye() command**.

So I stands for this is maybe a pun on the word identity. It's server set eye 4.

This is the 4 by 4 identity matrix. So I equals eye 4.

This gives me a 4 by 4 identity matrix.



And I equals eye 5, **eye 6**.

That gives me a 6 by 6 identity matrix,  $i_3$  is the 3 by 3 identity matrix.

```
>> I = eye(6)
I =
Diagonal Matrix
1 0 0 0 0 0
0 1 0 0 0 0
0 0 1 0 0 0
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1
```

## help() command

Lastly, to wrap up this video, there's one more useful command.

Which is the help command.

So you can type help eye and this brings up the help function for the identity matrix.

```
>>
>> help eye
```

```
eye' is a built-in function

-- Built-in Function: eye (X)
-- Built-in Function: eye (N, M)
-- Built-in Function: eye (... , CLASS)
    Return an identity matrix. If invoked with a single scalar
    argument, 'eye' returns a square matrix with the dimension
    specified. If you supply two scalar arguments, 'eye' takes them
    to be the number of rows and columns. If given a vector with two
    elements, 'eye' uses the values of the elements as the number of
    rows and columns, respectively. For example,
    eye (3)
        => 1 0 0
            0 1 0
            0 0 1
The following expressions all produce the same result:
    eye (2)
    ==
    eye (2, 2)
```

lines 1-22 -- (f)orward, (b)ack, (q)uit

**Hit Q to quit.**

And you can also type help rand. Brings up documentation for the rand or the random number generation function. Or even help help, which shows you, you know help on the help function.

```
>> help eye
>> help rand
>> help help
>>
```

So, those are the basic operations in Octave.

And with this you should be able to generate a few matrices, multiply, add things. And use the basic operations in Octave.

In the next video, I'd like to start talking about more sophisticated commands and how to use data around and start to process data in Octave.

## How to move Data

In this second tutorial video on Octave, I'd like to start to tell you how to move data around in Octave. So, if you have data for a machine learning problem, how do you load that data in Octave? How do you put it into matrix? How do you manipulate these matrices? How do you save the results? How do you move data around and operate with data?

### Recall

Here's my Octave window as before, picking up from where we left off in the last video. If I type A, that's the matrix so we generate it, right, with this command equals one, two, three, four, five, six, and this is a three by two matrix.

```
>> help eye
>> help rand
>> help help
>>
>>
>>
>>
>>
>> A
A =
    1   2
    3   4
    5   6
>> A = [1 2; 3 4; 5 6]
A =
    1   2
    3   4
    5   6
```

### Size command

The size command in Octave lets you, tells you what the size of a matrix is. So size A returns three, two. It turns out that this size command itself is actually returning a one by two matrix.

```
>> size(A)
ans =
    3   2
```

So you can actually set SZ equals size of A and SZ is now a one by two matrix where the first element of this is three, and the second element of this is two. So, if you just type size of SZ. Does SZ is a one by two matrix whose two elements contain the dimensions of the matrix A.

```
>> sz = size(A)
sz =
    3    2
>>
```

You can also type size A one to give you back the first dimension of A, size of the first dimension of A. So that's the number of rows and size A two to give you back two, which is the number of columns in the matrix A.

```
>>
>> size(A,1)
ans = 3
>> size(A,2)
ans = 2
>>
```

If you have a vector V, so let's say V equals one, two, three, four, and you type length V. What this does is it gives you the size of the longest dimension.

```
>> v = [1 2 3 4]
v =
    1    2    3    4
>> length(v)
ans = 4
>>
```

So you can also type length A and because A is a three by two matrix, the longer dimension is of size three, so this should print out three.

```
>>
>> length(A)
ans = 3
>>
```

But usually we apply length only to vectors. So you know, length one, two, three, four, five, rather than apply length to matrices because that's a little more confusing.

```
>> length([1;2;3;4;5])
ans = 5
>>
```

Now, let's look at how the load data and find data on the file system.

## Load and Find Data

### PWD command

When we start an Octave we're usually, we're often in a path that is, you know, the location of where the Octave location is. So the PWD command shows the current directory, or the current path that Octave is in. So right now we're in this maybe somewhat off scale directory.

```
>>
>> pwd
ans = C:\Octave\3.2.4_gcc-4.4.0\bin
>>
```

## CD command

The CD command stands for change directory, so I can go to C:/Users/Ang/Desktop, and now I'm in, you know, in my Desktop.

```
>>
>> pwd
ans = C:\Octave\3.2.4\gcc-4.4.0\bin
>>
>> cd 'C:\Users\ang\Desktop'
>> pwd
ans = C:\Users\ang\Desktop
>>
```

## LS command

If I type ls, ls is, it comes from a Unix or a Linux command. But, ls will list the directories on my desktop and so these are the files that are on my Desktop right now.

```
>> ls
Volume in drive C has no label.
Volume Serial Number is 0C32-E0EC

Directory of C:\Users\ang\Desktop

[.] [lectures-slides] squareThisNumber.m
[..] matlab_session.m
costFunctionJ.m [ml-class-ex1]
featuresX.dat priceY.dat
      5 File(s)    8,071 bytes
      4 Dir(s)  408,465,044,480 bytes free
>>
```

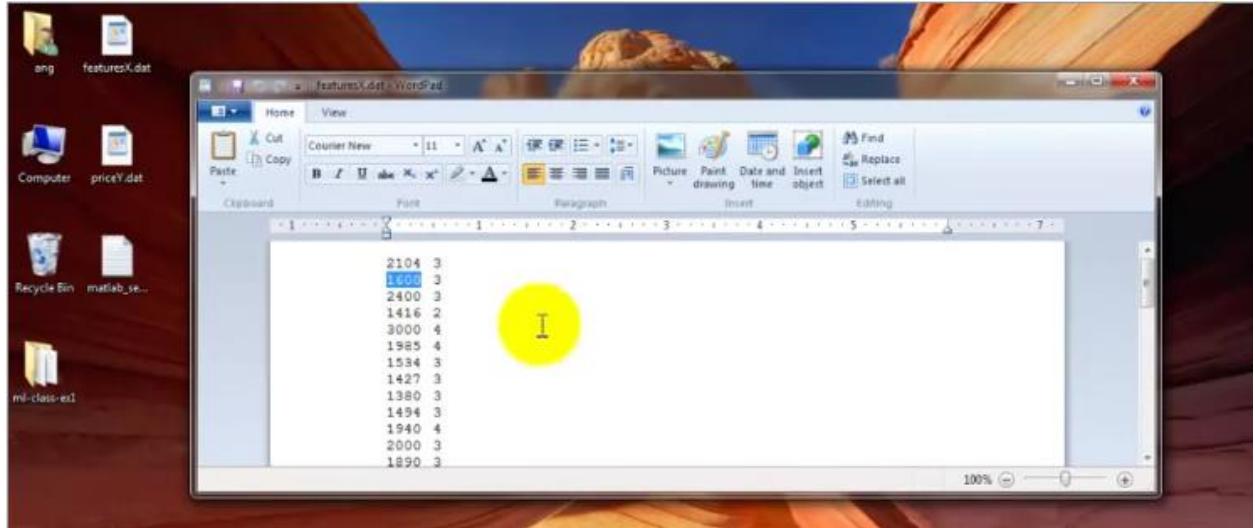
## featuresx and pricey files

In fact, on my desktop are two files: **featuresx** and **pricey** that's maybe come from a machine learning problem I want to solve. So, here's my desktop. Here's featuresx.



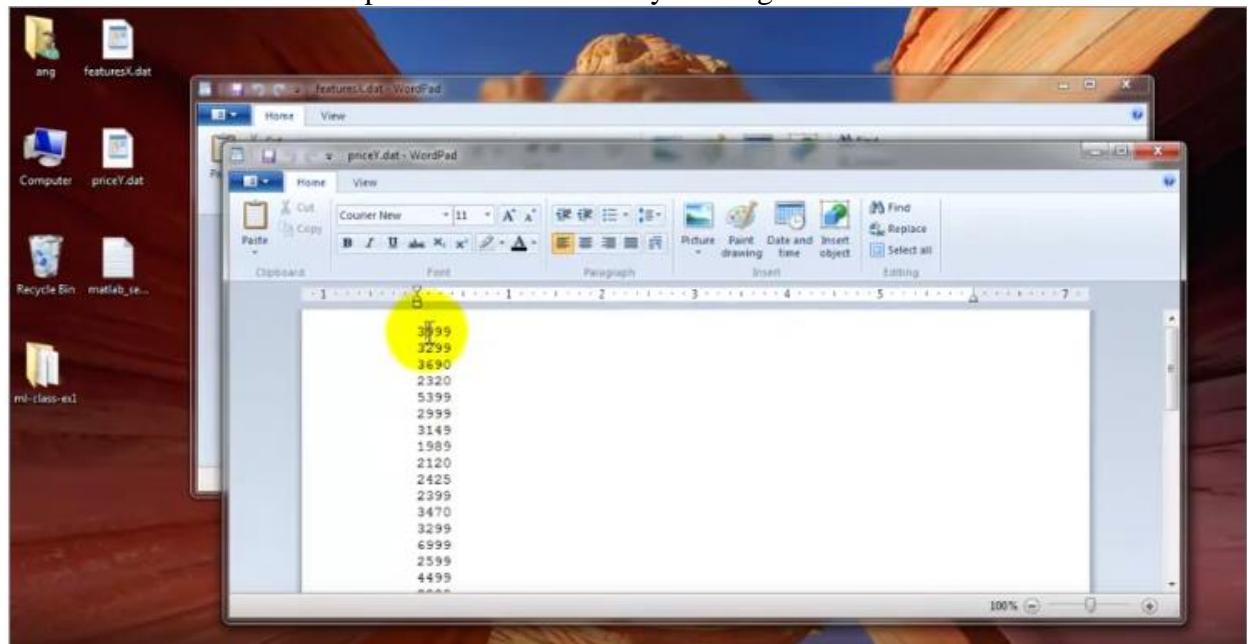
## featuresx file

And featuresx is this window, excuse me, is this file with two columns of data. This is actually my housing prices data. So I think, you know, I think I have forty-seven rows in this data set. And so the first house has size two hundred four square feet, has three bedrooms; second house has sixteen hundred square feet, has three bedrooms; and so on.



## pricey file

And Price Y is this file that has the prices of the data in my training set.



So, Features X and Price Y are just **text files with my data**.

## Load File

How do I load this data into Octave? Well, I just type the command `load featuresx.dat` and if I do that, I load the `featuresx` and can load `priceY.dat`. And by the way, there are multiple ways to do this. This command if you put `Features X.dat` on that in strings and load it like so. This is a typo there. This is an equivalent command. So you can, this way I'm just putting the file name of the string in the finding in a string and in an Octave use single quotes to represent strings, like so. So that's a string, and we can load the file whose name is given by that string.

```
>>
>> load featuresX.dat
>> load priceY.dat
>> load('featureX.dat')
error: load: unable to find file featureX.dat
>> load('featuresX.dat')
>> load('featuresX.dat')
>>
```

## who command

Now the WHO command now shows me what variables I have in my Octave workspace. So **who** shows me whether the variables that Octave has in memory currently? Features X and Price Y are among them, as well as the variables that, you know, we created earlier in this session.

```
>>
>> who
Variables in the current scope:
A           I           ans          c   featuresX  priceY      v
C           a           b           sz            w
>>
```

So I can type `Features X` to display `features X`. And there's my data.

```
>> featuresX
```

```

featuresX =
2104      3
1600      3
2400      3
1416      2
3000      4
1985      4
1534      3
1427      3
1380      3
1494      3
1940      4
2000      3
1890      3
4478      5
1268      3
2300      4
1320      2
1236      3
2609      4
3031      4

```

lines 1-22 -- (f)orward, (b)ack, (q)uit

And I can type size **featuresX** and that's my 47 by two matrix.

```

>> size(featuresX)
ans =
    47      2

```

And size **priceY**, that gives me my 47 by one vector. This is a 47 dimensional vector. This is all common vector that has all the prices Y in my training set.

```

>> size(priceY)
ans =
    47      1

```

Now the **who** function shows you one of the variables that, in the current workspace.

```

>> who
Variables in the current scope:
A          I          ans          c          priceY      v
C          a          b          featuresX  sz          w

```

## whos command

There's also the who S variable that gives you the **detailed view**. And so this also, with an S at the end this also lists my variables except that it now lists the **sizes as well**. So A is a three by

two matrix and featuresx as a 47 by 2 matrix. pricey is a 47 by one matrix. Meaning this is just a vector. And it shows, you know, how many bytes of memory it's taking up. As well as what type of data this is. **Double** means double position floating point so that just means that these are real values, the floating point numbers.

```
>> whos
Variables in the current scope:

Attr Name          Size            Bytes  Class
==== ============ ====== =====
A                3x2             48    double
C                2x3             48    double
I                6x6             48    double
a                1x1              8    double
ans               1x2             16    double
b                1x2              2    char
c                1x1              1    logical
featuresX        47x2            752   double
priceY           47x1            376   double
sz                1x2             16    double
v                1x4             32    double
w                1x10000          80000  double

Total is 10201 elements using 81347 bytes
>>
```

## clear command

Now if you want to get rid of a variable you can use the clear command. So clear featuresx and type whose again. You notice that the **featuresx** variable has now disappeared.

```
>> clear featuresX
>> whos
Variables in the current scope:

Attr Name          Size            Bytes  Class
==== ============ ====== =====
A                3x2             48    double
C                2x3             48    double
I                6x6             48    double
a                1x1              8    double
ans               1x2             16    double
b                1x2              2    char
c                1x1              1    logical
priceY           47x1            376   double
sz                1x2             16    double
v                1x4             32    double
w                1x10000          80000  double

Total is 10107 elements using 80595 bytes
>>
```

## Save Data

And **how do we save data?** Let's see. Let's take the variable V and say that it's a priceY 1 colon 10. This sets V to be the first 10 elements of vector Y.

```
>>
>> v = priceY(1:10)
v =
3999
3299
3690
2320
5399      ↗
2999
3149
1989
2120
2425

>>
```

So let's type who or whose.

```
>> who
Variables in the current scope:
A          I          ans         c       sz          w
C          a          b          priceY    v

>>
```

Whereas Y was a 47 by 1 vector. V is now 10 by 1.

```
>> whos
Variables in the current scope:

Attr Name            Size           Bytes  Class
==== ==             =====          =====
A          3x2            48   double
C          2x3            48   double
I          6x6            48   double
a          1x1             8   double
ans        1x2            16   double
b          1x2             2   char
c          1x1             1   logical
priceY    47x1          376   double
sz          1x2            16   double
v          10x1           80   double
w          1x10000        80000  double

Total is 10113 elements using 80643 bytes

>>
```

v equals priceY, one column ten that sets it to the just the first ten elements of Y.

```
>> v = priceY(1:10)
v =
3999
3299
3690
2320
5399
2999
3149
1989
2120
2425
>>
```

Let's say I wanna save this to date to disc the command **save hello.mat V**. This will save the variable V into a file called hello.mat. So let's do that.

```
>>
>>
>> save hello.mat v;
>>
```

And now a file has appeared on my Desktop, you know, called Hello.mat.



**Note:** I happen to have MATLAB installed in this window, which is why, you know, this icon looks like this because Windows is recognized as it's a MATLAB file, but don't worry about it if this file looks like it has a different icon on your machine.

## clear command

And let's say I clear all my variables. So, if you type clear without anything then this actually deletes all of the variables in your workspace. So there's now nothing left in the workspace.

```
>> clear
>> whos
>> who
>>
```

## load command

And if I **load hello.mat**, I can now **load back my variable v**, which is the data that I previously saved into the hello.mat file.

```
>> load hello.mat
>> whos
Variables in the current scope:

  Attr Name          Size          Bytes  Class
  ===== =========      =====      ===== 
        v            10x1           80   double

Total is 10 elements using 80 bytes
>>
```

```
>> v
v =
3999
3299
3690
2320
5399
2999
3149
1989
2120
2425
>> |
```

## Save in Binary and Text Formats

So, hello.mat, what we did just now to save hello.mat to view, this save the data in a binary format, a somewhat **more compressed binary format**. So if v is a lot of data, this, you know, will be somewhat more compressing. Will take up less space. If you want to save your data in a human readable format then you type save hello.txt the variable v and then -ascii. So, this will **save it as a text or as ascii format** of text.

```
>> save hello.txt v -ascii    % save as text (ASCII)
>> |
```

And now, once I've done that, I have this file. Hello.txt has just appeared on my desktop, and if I open this up, we see that this is a text file with my data saved away.



So that's how you load and save data.

## Manipulate Data

Now let's talk a bit about how to manipulate data. Let's set  $A$  equals to that matrix again so  $A$  is my three by two matrix. So as indexing. So type  $A(3, 2)$ . This indexes into the 3, 2 elements of the matrix  $A$ . So, this is what, you know, in normally, we will write this as a subscript 3, 2 or  $A$  subscript, you know, 3, 2 and so that's the element and third row and second column of  $A$  which is the element of six.

```
>>
>> A =[1 2; 3 4; 5 6]
A =
1 2
3 4
5 6
>> A(3,2)
ans = 6
```

I can also type  $A$  two comma colon to fetch everything in the second row. So, the colon means every element along that row or column. So,  $A$  two comma colon is this second row of  $A$ . Right.

```

>> A =[1 2; 3 4; 5 6]
A =
1 2
3 4
5 6

>> A(3,2)
ans = 6
>> A(2,:)
ans = % ":" means every element along that row/column
      3 4
>>

```

And similarly, if I do A colon comma 2 then this means get everything in the second column of A. So, this gives me 2 4 6. Right this means of A. everything, second column. So, this is my second column A, which is 2 4 6.

```

>> A(:,2)
ans =
2
4
6
>>

```

## More Sophisticated Commands

Now, you can also use somewhat most of the **sophisticated index in the operations**. So, we just click each of an example. You do this maybe less often, but let me do this A 1 3 comma colon. This means get all of the elements of A who's first indexes one or three. This means I **get everything from the first and third rows of A and from all columns**. So, this was the matrix A and so A 1 3 comma colon means get everything from the first row and from the third row and the colon means, you know, on both of first and the second columns and so this gives me this 1 2 5 6.

<pre> &gt;&gt; A([1 3], :) ans = 1 2 5 6 </pre>	<pre> &gt;&gt; A A = 1 2 3 4 5 6 </pre>
---	---

Although, you use these sorts of more sophisticated index operations, maybe **somewhat less often**.

To show you what else we can do. Here's the A matrix and this A colon two gives me the second column.

```
>> A(:,2)
ans =
    2
    4
    6
```

You can also use this to do assignments. So I can take the second column of A and assign that to 10, 11, 12, and if I do that I'm now, you know, taking the second column of A and I'm assigning this column vector 10, 11, 12 to it. So, now A is this matrix that's 1, 3, 5. And the **second column has been replaced** by 10, 11, 12.

```
>>
>> A
A =
    1    2
    3    4
    5    6

>> A(:,2)
ans =
    2
    4
    6

>> A(:,2) = [10; 11; 12]
A =
    1    10
    3    11
    5    12
```

And here's another operation. Let's set A to be equal to A comma 100, 101, 102 like so and what this will do is **append another column vector to the right**. So, now, oops. I think I made a little mistake. Should have put semicolons there and now A is equals to this. Okay? I hope that makes sense. So this 100, 101, 102. This is a column vector. And then we put that column vector to the right and so, we ended up taking the matrix A and--which was these six elements on the left. So we took matrix A and we appended another column vector to the right; which is now why A is a three by three matrix that looks like that.

```
>> A = [A, [100, 101, 102]]; % append another column vector to right
error: number of rows must match (1 != 3)
>> A = [A, [100; 101; 102]]; % append another column vector to right
>> A
A =
    1    10    100
    3    11    101
    5    12    102
```

And finally, one neat trick that I sometimes use if you do just a and just a colon like so. This is a somewhat special case syntax. What this means is that put all elements with A into a single

column vector and this gives me a 9 by 1 vector. They adjust the other ones are combined together.

```
>> A(:) % put all elements of A into a single vector
ans =
1
3
5
10
11
12
100
101
102
```

Just a couple more examples. Let's see. Let's say I set A to be equal to 123456, okay? And let's say I set a B to B equal to 11, 12, 13, 14, 15, 16.

```
>> A = [1 2; 3 4; 5 6];
>> B = [11 12; 13 14; 15 16]
B =
11 12
13 14
15 16
```

I can create a new matrix C as A B. This just means my Matrix A. Here's my Matrix B and I've set C to be equal to AB. What I'm doing is I'm taking these two matrices and just concatenating onto each other. So the left, matrix A on the left. And I have the matrix B on the right. And that's how I formed this matrix C by putting them together.

<pre>&gt;&gt; A A =</pre>	<pre>&gt;&gt; B B =</pre>	<pre>&gt;&gt; C = [A B] C =</pre>
$\begin{matrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{matrix}$	$\begin{matrix} 11 & 12 \\ 13 & 14 \\ 15 & 16 \end{matrix}$	$\begin{matrix} 1 & 2 & 11 & 12 \\ 3 & 4 & 13 & 14 \\ 5 & 6 & 15 & 16 \end{matrix}$

I can also do C equals A semicolon B. The semi colon notation means that I go put the next thing at the bottom. So, I'll do is a equals semicolon B. It also puts the matrices A and B together except that it now puts them on top of each other. so now I have A on top and B at the bottom and C here is now in 6 by 2 matrix. So, just say the **semicolon thing usually means, you know, go to the next line**. So, C is comprised by A and then go to the bottom of that and then put B in the bottom.

```
>> C = [A; B]
C =
1   2
3   4
5   6
11  12
13  14
15  16
```

And by the way, this **A B** is the same as **A, B** and so you know, either of these gives you the same result.

```
>> [A B]
ans =
1   2   11   12
3   4   13   14
5   6   15   16

>> [A, B]
ans =
1   2   11   12
3   4   13   14
5   6   15   16
```

So, with that, hopefully you now know how to construct matrices and hopefully starts to show you some of the commands that you use to quickly put together matrices and take matrices and, you know, slam them together to form bigger matrices, and with just a few lines of code, Octave is very convenient in terms of how quickly we can assemble complex matrices and move data around. So that's it for moving data around. In the next video we'll start to talk about how to actually do complex computations on this, on our data. So, hopefully that gives you a sense of how, with just a few commands, you can very quickly move data around in Octave. You know, you load and save vectors and matrices, load and save data, put together matrices to create bigger matrices, index into or select specific elements on the matrices. I know I went through a lot of commands, so I think the best thing for you to do is afterward, to look at the transcript of the things I was typing. You know, look at it. Look at the coursework site and download the transcript of the session from there and look through the transcript and type some of those commands into Octave yourself and start to play with these commands and get it to work. And obviously, you know, there's no point at all to try to memorize all these commands. It's just, but what you should do is, hopefully from this video you have gotten a sense of the sorts of things you can do. So that when later on when you are trying to program a learning algorithms yourself, if you are trying to find a specific command that maybe you think Octave can do because you think you might have seen it here, you should refer to the transcript of the session and look through that in order to find the commands you wanna use.

So, that's it for moving data around and **in the next video what I'd like to do is start to tell you how to actually do complex computations** on our data, and how to compute on the data, and actually start to implement learning algorithms.

## Computing on Data

Now that you know how to load and save data in Octave, put your data into matrices and so on. In this video, I'd like to show you how to do **computational operations on data**. And later on, we'll be using these source of **computational operations to implement our learning algorithms**.

Let's get started

```
octave-3.2.4.exe:1> PS1('>> ')  
>>  
>>
```

### Declaring Matrices

Here's my Octave window. Let me just quickly initialize some variables to use for our example. So set A to be a three by two matrix, and set B to a three by two matrix, and let's set C to a two by two matrix like so.

```
>>  
>> A = [1 2; 3 4; 5 6]  
A =  
  
    1    2  
    3    4  
    5    6  
  
>> B = [11 12; 13 14; 15 16]  
B =  
  
   11    12  
   13    14  
   15    16  
  
>> C = [1 1; 2 2]  
C =  
  
    1    1  
    2    2
```

```
>>
```

## Multiplying Matrices

Now let's say I want to multiply two of my matrices. So let's say I want to compute  $A^*C$ , I just type  $A^*C$ , so it's a three by two matrix times a two by two matrix, this gives me this three by two matrix.

```
>> A*C
ans =
      5     5
     11    11
     17    17
>>
```

## Multiplying Matrices Element Wise

You can also do element wise operations and do  $A.^* B$  and what this will do is it'll take each element of A and multiply it by the corresponding elements B, so that's A, that's B, that's  $A.^* B$ . So for example, the first element gives 1 times 11, which gives 11. The second element gives 2 times 12 which gives 24, and so on. So this is element-wise multiplication of two matrices. And in general, the **period is usually used to denote element-wise operations** in Octave.

```
>> A .* B
ans =
      11    24
      39    56
      75    96
```

## Squaring Element Wise

So here's a matrix A, and if I do  $A.^2$ , this gives me the element wise squaring of A. So 1 squared is 1, 2 squared is 4, and so on.

```
>> A.^2
ans =
      1     4
      9    16
     25    36
```

## Reciprocal (Inverse) of Vector or Matrices Element Wise

Let's set v as a vector. Let's set v as one, two, three as a column vector. You can also do one dot over v to do the element-wise reciprocal of v, so this gives me one over one, one over two, and one over three.

```
>> v = [1; 2; 3]
v =
1
2
3
>> 1 ./ v
ans =
1.00000
0.50000
0.33333
>>
```

And this is where I do the **matrices**, so one dot over A gives me the element wise inverse of A. And once again, the period here gives us a clue that this an element-wise operation.

```
>> 1 ./ A
ans =
1.00000 0.50000
0.33333 0.25000
0.20000 0.16667
>>
```

We can also do things like **log(v)**, this is a element-wise logarithm of the v.

```
>> log(v)
ans =
0.00000
0.69315
1.09861
```

E to the V is **base E exponentiation of these elements**, so this is element squared e, because this was v,

```
>> exp(v)
ans =
2.7183
7.3891
20.0855
```

I can also do abs V to take the **element-wise absolute value of V**. So here, V was our positive.

```
>> abs(v)
ans =
1
2
3
```

abs, minus one, two minus 3, the element-wise absolute value gives me back these **non-negative values**.

```
>> abs([-1; 2;-3])
ans =
    1
    2
    3
```

And **negative v** gives me the minus of v. This is the same as negative one times v, but usually you just write negative v instead of  $-1^*v$ .

```
>> -v
ans =
    -1
    -2
    -3
```

And what else can you do? Here's another neat trick. So, let's see. Let's say I want to **take v and increment each of its elements by one**. Well one way to do it is by constructing a three by one vector that's all ones and adding that to v. So if I do that, this increments v by from 1, 2, 3 to 2, 3, 4. **The way I did** that was,  $\text{length}(v)$  is 3, so **ones(length(v),1)**, this is ones of 3 by 1, so that's ones(3,1) on the right and what I did was v plus ones, which is adding this vector of our ones to v, and so this increments v by one,

```
>> v
v =
    1
    2
    3

>> v + ones(length(v),1)
ans =
    2
    3
    4
```

Another **simpler way to do that is to type v plus one**. So here is v, and v plus one also means to add one element wise to each of my elements of v.

```
>> v + 1
ans =
    2
    3
    4
```

## Transpose

Now, let's talk about more operations. So here's my matrix A, if you want to buy A transposed, the way to do that is to write A prime, that's the **apostrophe symbol**, it's the left quote, so it's on

your keyboard, you have a left quote and a right quote. So this is actually the standard quotation mark. Just type A transpose, this gives me the transpose of my matrix A.

```
>> A
A =
    1   2
    3   4
    5   6

>> A'
ans =
    1   3   5
    2   4   6
```

And, of course, A transpose, if I transpose that again, then I should get back my matrix A.

```
>> (A')'
ans =
    1   2
    3   4
    5   6
```

Some more useful functions. Let's say lower case a is 1 15 2 0.5, so it's 1 by 4 matrix.

```
>> a = [1 15 2 0.5]
a =
    1.00000   15.00000   2.00000   0.50000
```

Let's say val equals max of a this returns the maximum value of a which in this case is 15.

```
>> val = max(a)
val = 15
>>
```

And I can do val, ind max(a) and this returns val and ind which are going to be the maximum value of A which is 15, as well as the index. So it was the element number two of A that was 15 so ind is my index into this.

```
>> [val, ind] = max(a)
val = 15
ind = 2
>>
```

Just as a warning, if you do max(A), where A is a matrix, what this does is this actually does the column wise maximum. But say a little more about this in a second.

## Element wise Comparison

Still using this example that there for lowercase a. If I do **a < 3**, this does the element wise operation. Element wise comparison, so the first element of A is less than three so this one. Second element of A is not less than three so this value says zero cuz it's false.

The third and fourth elements of A are less than three, so that's just 1. So that's the element-wise comparison of all four elements of the variable **a < 3**. And it returns true or false depending on whether or not there's less than three.

```
>> a < 3
ans =
1 0 1 1
>>
```

Now, if I do **find(a < 3)**, this will tell me which are the elements of a, the variable a, that are less than 3, and in this case, the first, third and fourth elements are less than 3.

```
>> find(a < 3)
ans =
1 3 4
>>
```

## magic function

For our next example, let me set A to be equal to **magic(3)**.

```
>> A = magic(3)
A =
8 1 6
3 5 7
4 9 2
>>
```

The magic function returns, let's type **help magic**.

```
Create an N-by-N magic square. Note that 'magic (2)' is undefined
since there is no 2-by-2 magic square.
```

The **magic function returns these matrices called magic squares**. They have this, you know, mathematical property that all of their rows and columns and diagonals sum up to the same thing. So, you know, it's not actually useful for machine learning as far as I know, but I'm just using this as a convenient way to generate a three by three matrix. And these magic squares have the property that each row, each column, and the diagonals all add up to the same thing, so it's kind of a mathematical construct. I use this magic function only when I'm doing demos or when I'm teaching octave like those in, I don't actually use it for any useful machine learning application.

## [r, c] find()

But let's see, if I type **RC = find(A > 7)** this finds All the elements of A that are greater than equal to seven, and **so r, c stands for row and column**. So the 1,1 element is greater than 7, the

3,2 element is greater than 7, and the 2,3 element is greater than 7. So let's see. The 2,3 element, for example, is A(2,3), is 7 is this element out here, and that is indeed greater than equal seven.

```
>> [r,c] = find(A >= 7)
r =
1
3
2
c =
1
2
3
```

## Using help

By the way, I actually don't even memorize myself what these find functions do and what all of these things do myself. And whenever I use the find function, sometimes I forget myself exactly what it does, and now I would type help find to look at the document.

## sum function

Okay, just two more things that I'll quickly show you. One is the sum function, so here's my a, and then type sum(a). This adds up all the elements of a,

```
>> a
a =
1.00000 15.00000 2.00000 0.50000
>> sum(a)
ans = 18.500
>>
```

## prod function

And if I want to multiply them together, I type prod(a) prod sends the product, and this returns the product of these four elements of A.

```
>> prod(a)
ans = 15
>>
```

## floor and ceil

Floor(a) rounds down these elements of A, so 0.5 gets rounded down to 0. And ceil, or ceiling(A) gets rounded up to the nearest integer, so 0.5 gets rounded up to 1.

```
>> floor(a)
ans =
    1    15    2    0
>> ceil(a)      *
ans =
    1    15    2    1
>>
```

## random and max random

You can also, let's see. Let me type `rand(3)`, this generates a three by three matrix. If I type `max(rand(3))`, what this does is it takes the element-wise maximum of 3 random 3 by 3 matrices. So you notice all of these numbers tend to be a bit on the large side because each of these is actually the max of a element wise max of two randomly generated matrices.

```
>> rand(3)
ans =
    0.8172101    0.7629192    0.5765014
    0.8586035    0.8683389    0.0034115
    0.6242835    0.9279313    0.7502126
>> max(rand(3), rand(3))
ans =
    0.72763    0.78773    0.93872
    0.72363    0.83590    0.42763
    0.48315    0.41734    0.79961
>>
```

## Column wise Maximum

This is my magic number. This is my magic square, three by three A. Let's say I type `max A`, and then this will be `a, [], 1`, what this does is this takes the column wise maximum. So the max of the first column is 8, max of second column is 9, the max of the third column is 7. This 1 means to take the max among the first dimension of 8.

```
>> max(A, [], 1)
ans =
    8    9    7
>>
```

## Row wise Maximum

In contrast, if I were to type `max A`, this funny notation, two, then this takes the per row maximum. So the max of the first row is eight, max of second row is seven, max of the third

row is nine, and so this allows you to take maxes either per row or per column.

```
>> max(A,[],2)
ans =
    8
    7
    9
>>
```

And remember the default's to a column wise element. So if you want to find the maximum element in the entire matrix A, you can type `max(max(A))` like so, which is 9.

```
>> max(A)
ans =
    8    9    7
>> max(max(A))
ans = 9
```

Or you can turn A into a vector and type `max(A(:))` like so and this treats this as a vector and takes the max element of that vector.

```
>> A(:)
ans =
    8
    3
    4
    1
    5
    9
    6
    7
    2
>> max(A(:))
ans = 9
>>
```

Finally let's set A to be a 9 by 9 magic square. So remember the magic square has this property that every column and every row sums the same thing, and also the diagonals, so just a nine by nine matrix square.

```
>> A = magic(9)
A =

```

47	58	69	80	1	12	23	34	45
57	68	79	9	11	22	33	44	46
67	78	8	10	21	32	43	54	56
77	7	18	20	31	42	53	55	66
6	17	19	30	41	52	63	65	76
16	27	29	40	51	62	64	75	5
26	28	39	50	61	72	74	4	15
36	38	49	60	71	73	3	14	25
37	48	59	70	81	2	13	24	35

So let me just **sum(A, 1)**. So this does a **per column sum**, so we'll take each column of A and add them up and this is verified that indeed for a nine by nine matrix square, every column adds up to 369, adds up to the same thing.

```
>> sum(A,1)
ans =

```

369	369	369	369	369	369	369	369	369
-----	-----	-----	-----	-----	-----	-----	-----	-----

Now let's do the row wide sum. So the **sum(A,2)**, and this **sums up each row** of A, and indeed each row of A also sums up to 369.

```
>> sum(A,2)
ans =

```

369	369	369	369	369	369	369	369	369
-----	-----	-----	-----	-----	-----	-----	-----	-----

Now, let's **sum the diagonal elements of A** and make sure that also sums up to the same thing. So what I'm gonna do is **construct a nine by nine identity matrix**, that's eye nine.

```
>> eye(9)
ans =
Diagonal Matrix

1 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 1
```

And let me take A and construct, multiply A element wise, so here's my matrix A.

```
>> A
A =
47 58 69 80 1 12 23 34 45
57 68 79 9 11 22 33 44 46
67 78 8 10 21 32 43 54 56
77 7 18 20 31 42 53 55 66
6 17 19 30 41 52 63 65 76
16 27 29 40 51 62 64 75 5
26 28 39 50 61 72 74 4 15
36 38 49 60 71 73 3 14 25
37 48 59 70 81 2 13 24 35
```

I'm going to do **A.^ eye(9)**. What this will do is take the **element wise product of these two matrices**, and so this should wipe out everything in A, except for the diagonal entries.

```
>> A .* eye(9)
ans =
47 0 0 0 0 0 0 0 0
0 68 0 0 0 0 0 0 0
0 0 8 0 0 0 0 0 0
0 0 0 20 0 0 0 0 0
0 0 0 0 41 0 0 0 0
0 0 0 0 0 62 0 0 0
0 0 0 0 0 0 74 0 0
0 0 0 0 0 0 0 14 0
0 0 0 0 0 0 0 0 35
```

And now, I'm gonna do sum sum of A of that and this gives me the sum of these diagonal elements, and indeed that is 369.

```
>> sum(sum(A.*eye(9)))
ans = 369
>>
```

You can sum up the other diagonals as well. So this top left to bottom left, you can sum up the opposite diagonal from bottom left to top right. The commands for this is somewhat more cryptic, you don't really need to know this. I'm just showing you this in case any of you are curious. But let's see. **flipud** stands for flip up down. But if you do that, that turns out to sum up the elements in the opposite. So the other diagram, that also sums up to 369. Here, let me show you.

```
>> sum(sum(A.*flipud(eye(9))))  
ans = 369  
>>
```

Whereas `eye(9)` is this matrix.

```
>> eye(9)  
ans =  
  
Diagonal Matrix  
  
1 0 0 0 0 0 0 0 0  
0 1 0 0 0 0 0 0 0  
0 0 1 0 0 0 0 0 0  
0 0 0 1 0 0 0 0 0  
0 0 0 0 1 0 0 0 0  
0 0 0 0 0 1 0 0 0  
0 0 0 0 0 0 1 0 0  
0 0 0 0 0 0 0 1 0  
0 0 0 0 0 0 0 0 1
```

**flipud(eye(9))**, takes the identity matrix, and flips it vertically, so you end up with, excuse me, flipud, end up with ones on this opposite diagonal as well.

```
>> flipud(eye(9))  
ans =  
  
Permutation Matrix  
  
0 0 0 0 0 0 0 0 1  
0 0 0 0 0 0 0 1 0  
0 0 0 0 0 0 1 0 0  
0 0 0 0 0 1 0 0 0  
0 0 0 0 1 0 0 0 0  
0 0 0 1 0 0 0 0 0  
0 0 1 0 0 0 0 0 0  
0 1 0 0 0 0 0 0 0  
1 0 0 0 0 0 0 0 0
```

Just one last command and then that's it, and then that'll be it for this video. Let's set A to be the three by three magic square. If you want to invert a matrix, you type **pinv(A)**. This is typically called the **pseudo-inverse**, but it doesn't matter. Just think of it as basically the inverse of A, and that's the inverse of A.

```
>> A = magic(3)
A =
8   1   6
3   5   7
4   9   2

>> pinv(A)
ans =
0.147222  -0.144444  0.063889
-0.061111  0.022222  0.105556
-0.019444  0.188889  -0.102778
```

And so I can set **temp = pinv(A)** and **temp times A**, this is indeed the **identity matrix**, where it's essentially **ones on the diagonals**, and zeroes on the off-diagonals, up to a numeric round off.

```
>> temp = pinv(A)
temp =
0.147222  -0.144444  0.063889
-0.061111  0.022222  0.105556
-0.019444  0.188889  -0.102778

>> temp * A
ans =
1.0000e+000  1.5266e-016  -2.8588e-015
-6.1236e-015  1.0000e+000  6.2277e-015
3.1364e-015  -3.6429e-016  1.0000e+000
```

So, that's it for how to do different computational operations on data and matrices.

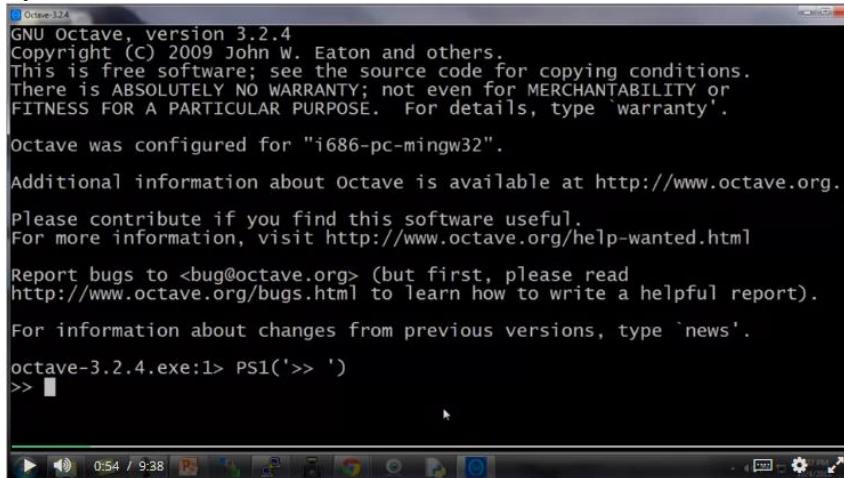
And **after running a learning algorithm**, often one of **the most useful things is to be able to look at your results, so to plot or visualize your result**. And in the next video, I'm going to very quickly show you how again with one or two lines of code using Octave. You can quickly **visualize your data or plot your data and use that to better understand what you're learning algorithms are doing**.

## Plotting Data

When developing learning algorithms, very often a few simple plots can give you a better sense of what the algorithm is doing and just sanity check that everything is going okay and the algorithms doing what is supposed to. For example, in an earlier video, I talked about how plotting the cost function  $J$  of theta can help you make sure that gradient descent is converging. Often, plots of the data or of all the learning algorithm outputs will also give you ideas for how to improve your learning algorithm. Fortunately, Octave has very simple tools to generate lots

of different plots and when I use learning algorithms, I find that plotting the data, plotting the learning algorithm and so on are often an important part of how I get ideas for improving the algorithms and in this video, I'd like to show you some of these Octave tools for plotting and visualizing your data.

Here's my Octave window.



```
Octave-3.2.4
GNU Octave, version 3.2.4
Copyright (c) 2009 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. For details, type 'warranty'.
Octave was configured for "i686-pc-mingw32".
Additional information about Octave is available at http://www.octave.org.
Please contribute if you find this software useful.
For more information, visit http://www.octave.org/help-wanted.html
Report bugs to <bug@octave.org> (but first, please read
http://www.octave.org/bugs.html to learn how to write a helpful report).
For information about changes from previous versions, type 'news'.
octave-3.2.4.exe:1> PS1('>> ')
>> 
```

Let's quickly generate some data for us to plot. So I'm going to set T to be equal to, you know, this array of numbers.

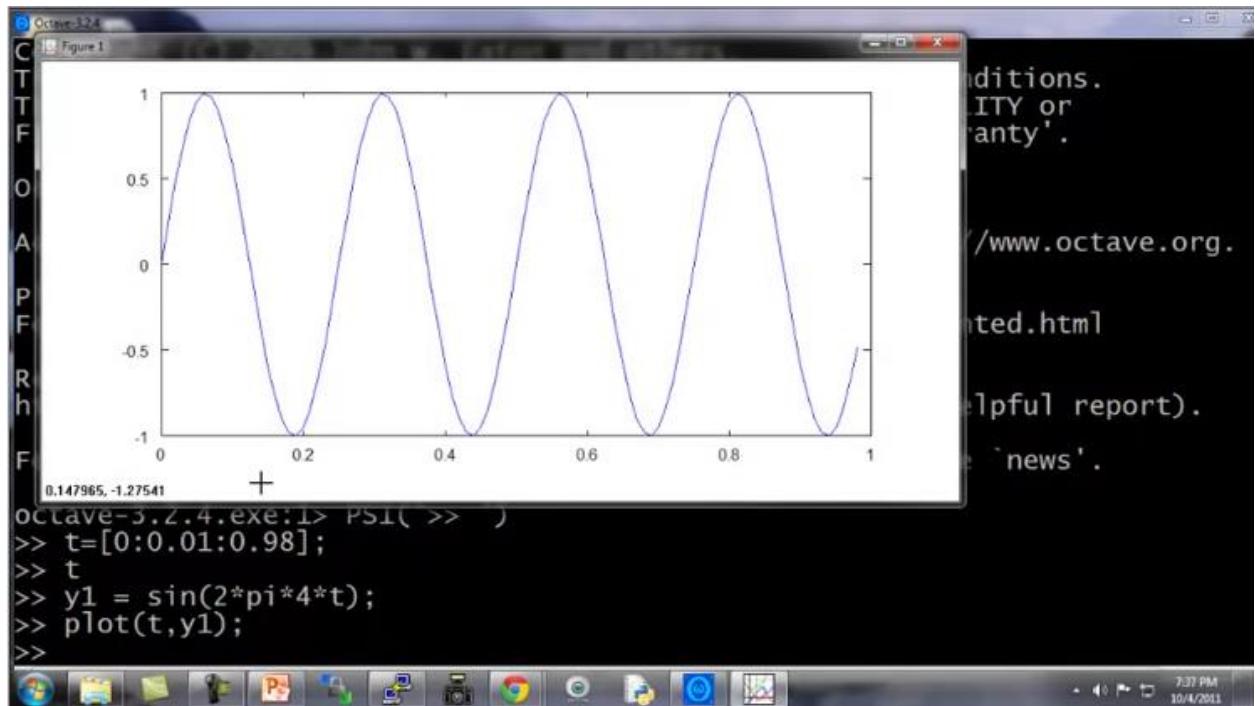
```
octave-3.2.4.exe:1> PS1('>> ')
>> t=[0:0.01:0.98];
>> t
```

Here's T, set of numbers going from 0 up to .98.

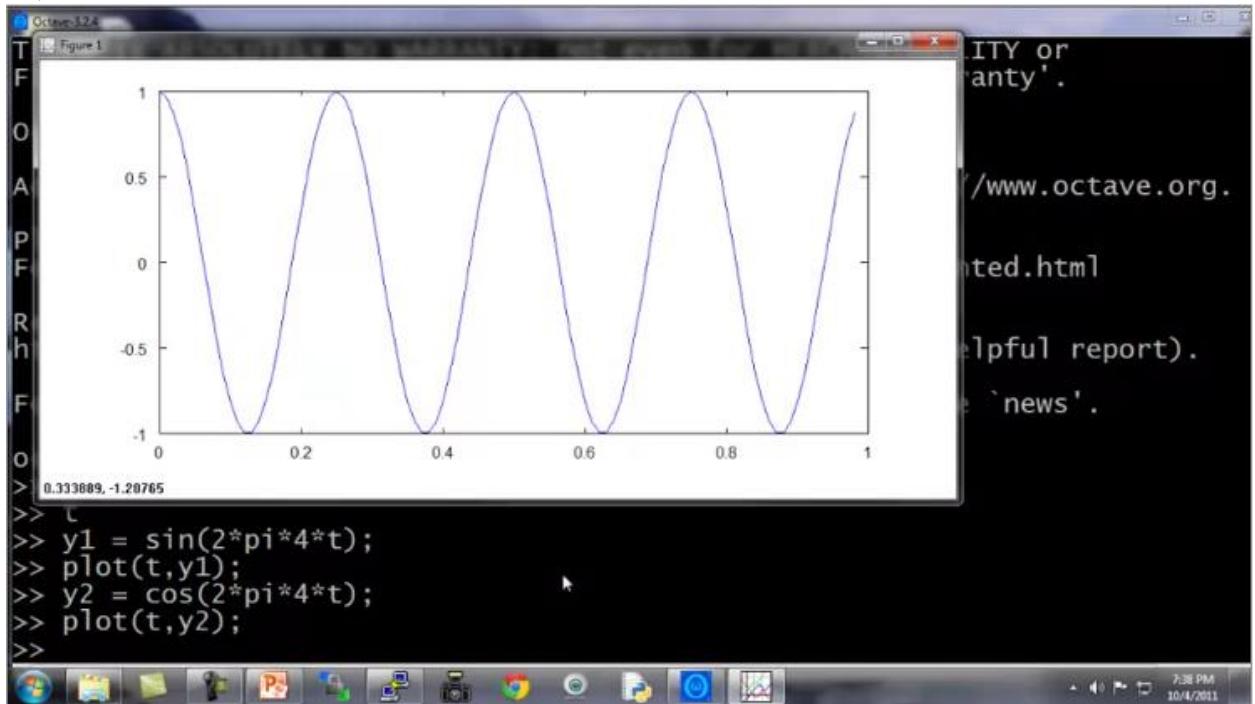
```
t =
Columns 1 through 7:
0.00000 0.01000 0.02000 0.03000 0.04000 0.05000 0.06000
Columns 8 through 14:
0.07000 0.08000 0.09000 0.10000 0.11000 0.12000 0.13000
Columns 15 through 21:
0.14000 0.15000 0.16000 0.17000 0.18000 0.19000 0.20000
Columns 22 through 28:
0.21000 0.22000 0.23000 0.24000 0.25000 0.26000 0.27000
Columns 29 through 35:
0.28000 0.29000 0.30000 0.31000 0.32000 0.33000 0.34000
Lines 1-22 -- (f)orward (b)ack (q)uit
```

Let's set y1 equals sine of 2 pi 4 T and if I want to plot the sine function, it's very easy. I just type plot T comma Y1 and hit enter. And up comes this plot where the horizontal axis is the T

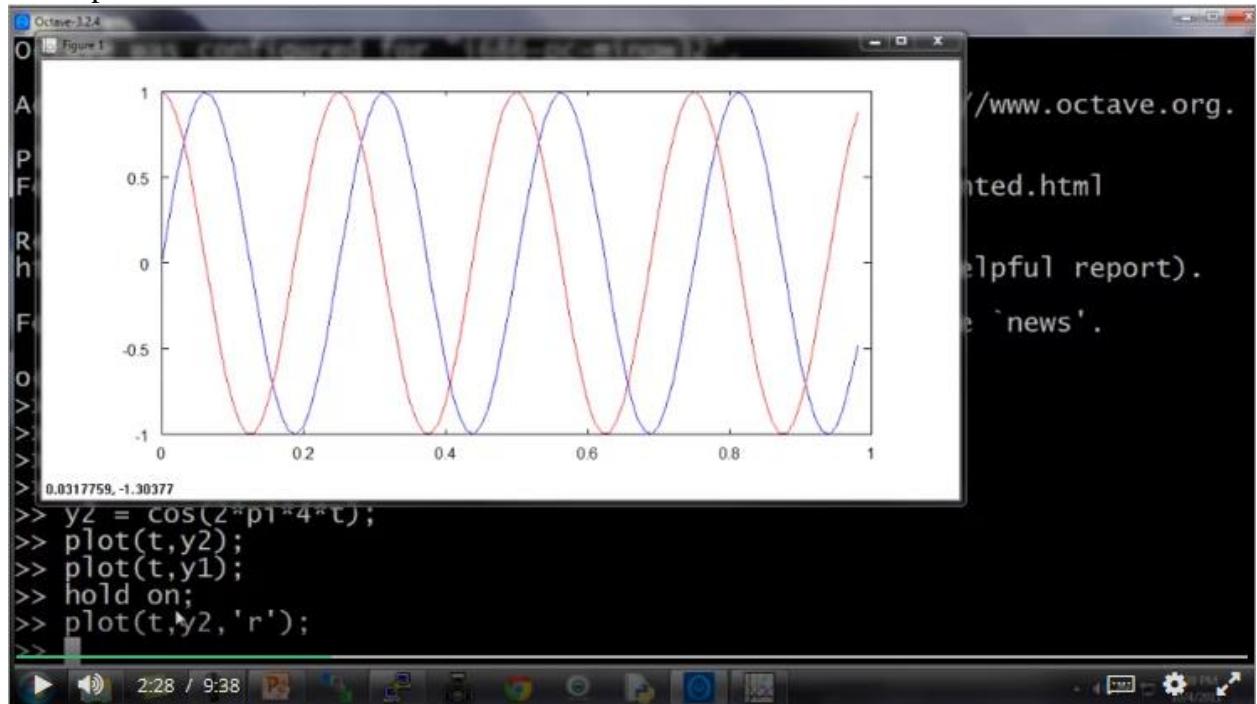
variable and the vertical axis is  $y_1$ , which is the sine you saw in the function that we just computed.



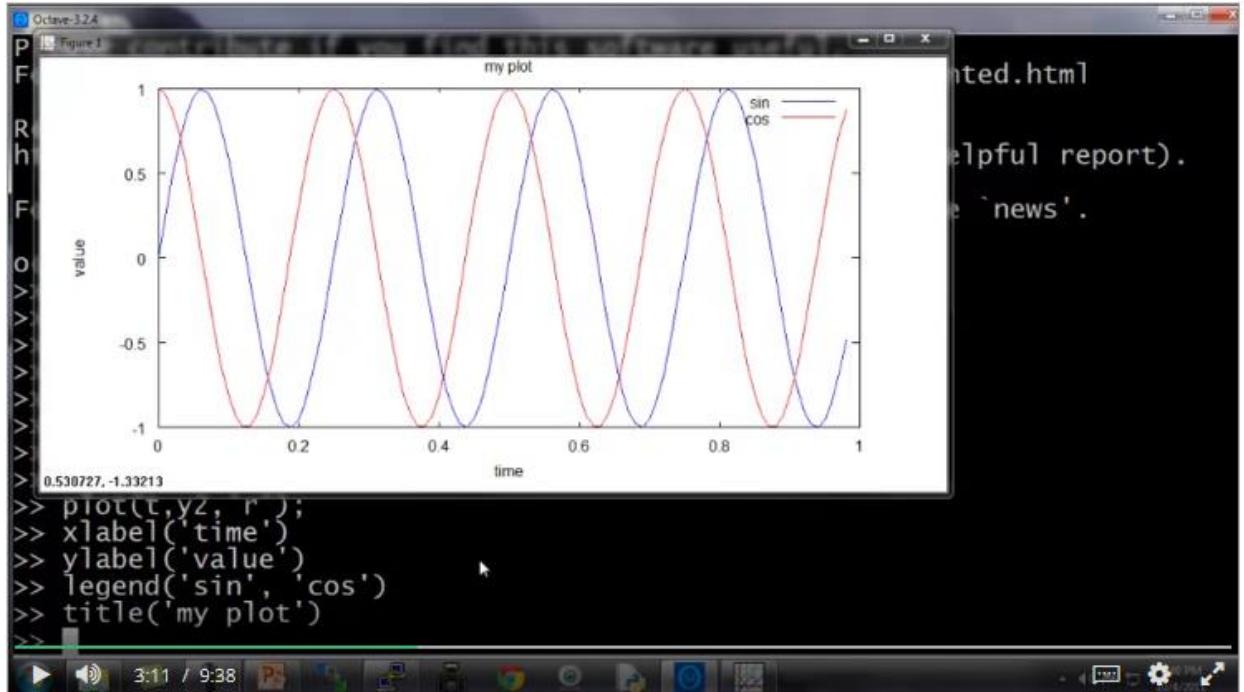
Let's set  $y_2$  to be equal to the **cosine of two pi four T**, like so. And if I plot T comma  $y_2$ , what octave will do is it will take my sine plot and it will replace with this cosine function and now, you know, cosine of xi of 1.



Now, what if I want to have both the sine and the cosine plots on top of each other? What I'm going to do is I'm going to type plot **t,y1**. So here's my sine function, and then I'm going to use the function **hold on**. And what hold does it closes octaves to now figures on top of the old one and let me now plot **t y2**. I'm going to plot the cosine function in a different color. So, let me put there **r** in quotation marks there and instead of replacing the current figure, I'll plot the cosine function on top and the r indicates what is an event color.



And here additional commands **- x label time**, to label the X axis, or the horizontal axis. And Y **label values** A, to label the vertical axis value, and I can also label my two lines with this command: **legend sine cosine** and this puts this legend up on the upper right showing what the 2 lines are, and finally **title my plot** is the title at the top of this figure.



Lastly, if you want to **save this figure**, you type print -dpng myplot.png. So PNG is a graphics file format, and if you do this **it will let you save this as a file**. If I do that, let me actually change directory to, let's see, like that, and then I will print that out. So this will take a while depending on how your Octave configuration is setup, may take a few seconds, but change directory to my desktop and Octave is now taking a few seconds to save this.

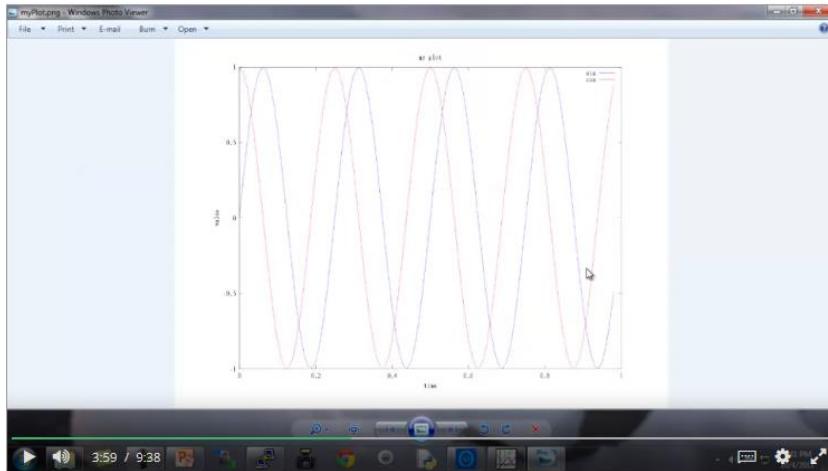
```

>> cd 'C:\Users\ang\Desktop'; print -dpng 'myPlot.png'
warning: implicit conversion from matrix to string

```

If I now go to my desktop, Let's hide these windows. **Here's myplot.png which Octave has saved**, and you know, there's the figure saved as the PNG file.





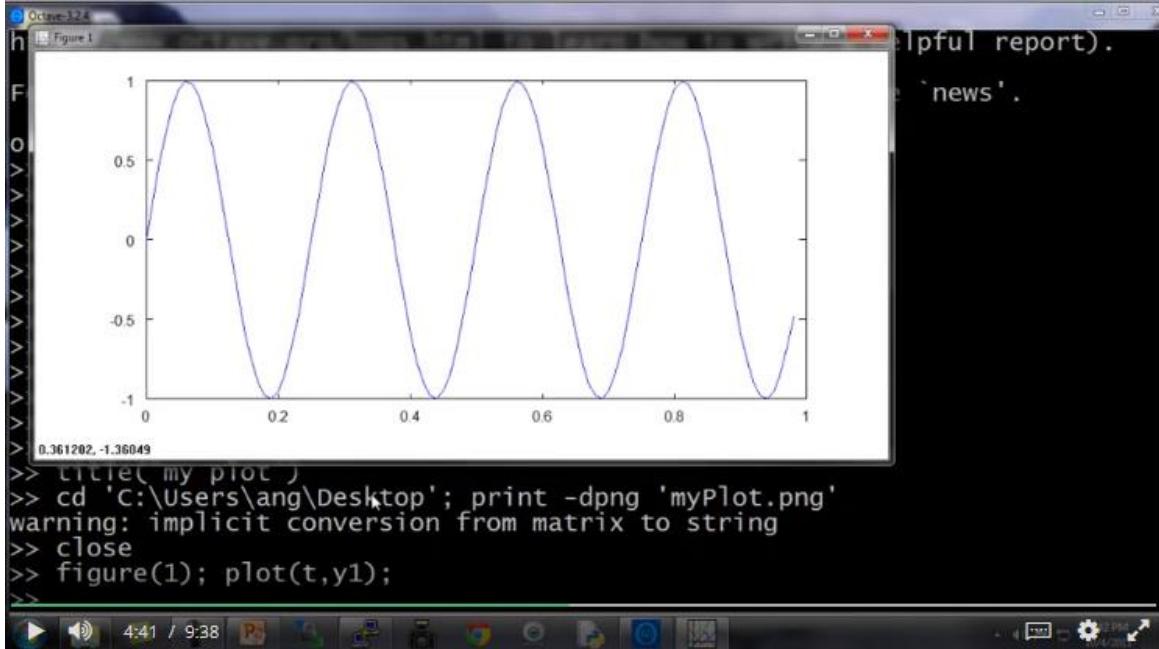
Octave can save thousand other formats as well. So, you can **type help plot**, if you want to see the other file formats, rather than PNG, that you can save figures in.

```
>> help plot
```

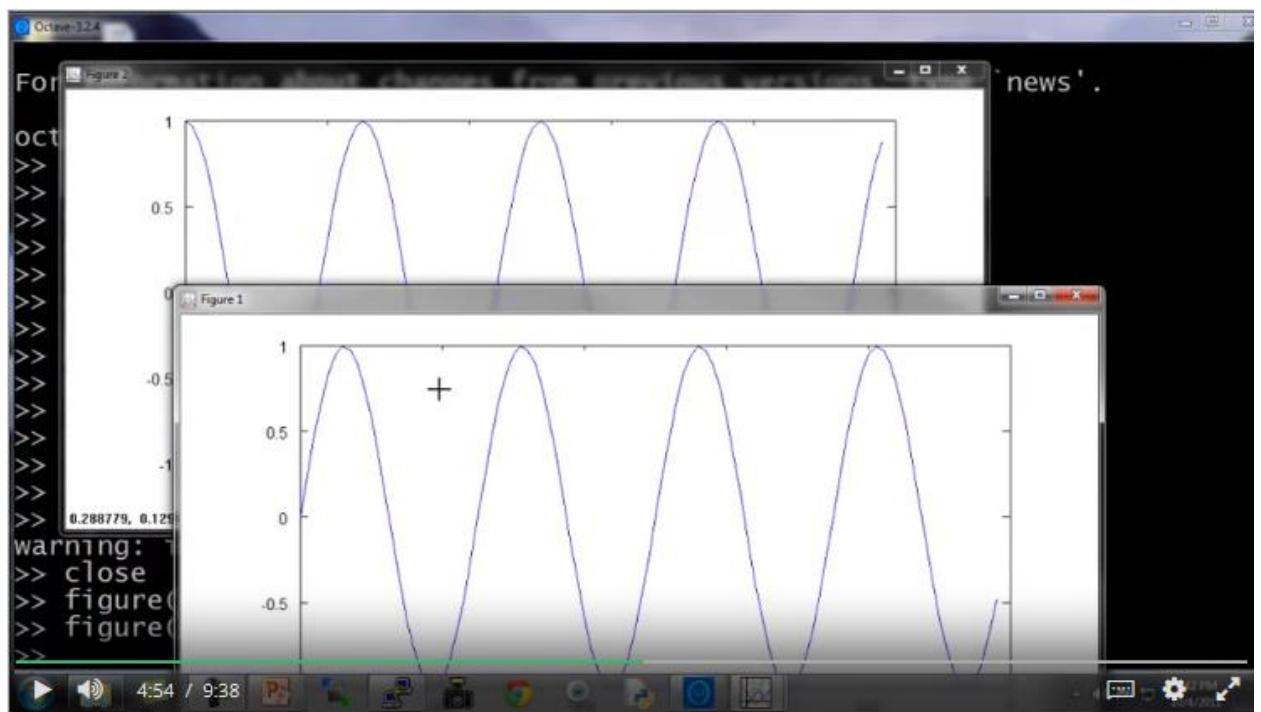
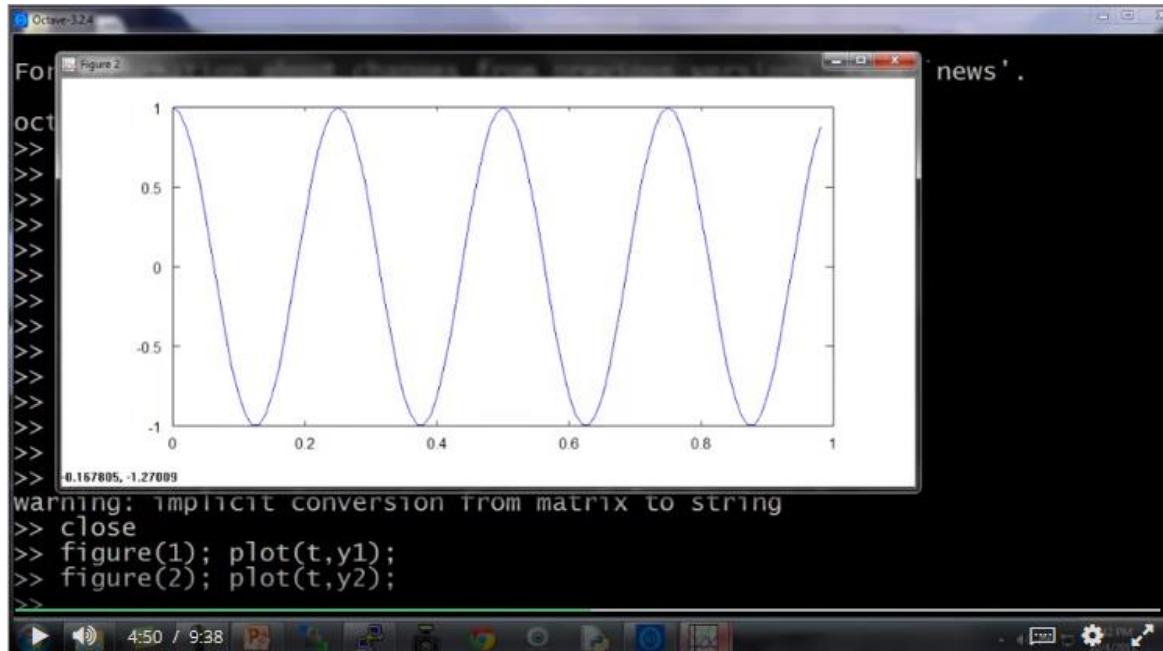
And lastly, if you want to get rid of the plot, the **close command** causes the figure to go away. As I figure if I type close, that figure just disappeared from my desktop.

```
>> close
```

Octave also lets you specify a figure and numbers. You **type figure 1 plots t, y1**. That starts up first figure, and that plots t, y1.



And then **if you want a second figure**, you specify a different figure number. So figure two, plot t, y2 like so, and now on my desktop, **I actually have 2 figures**. So, figure 1 and figure 2 thus 1 plotting the sine function, 1 plotting the cosine function.



### subplot command

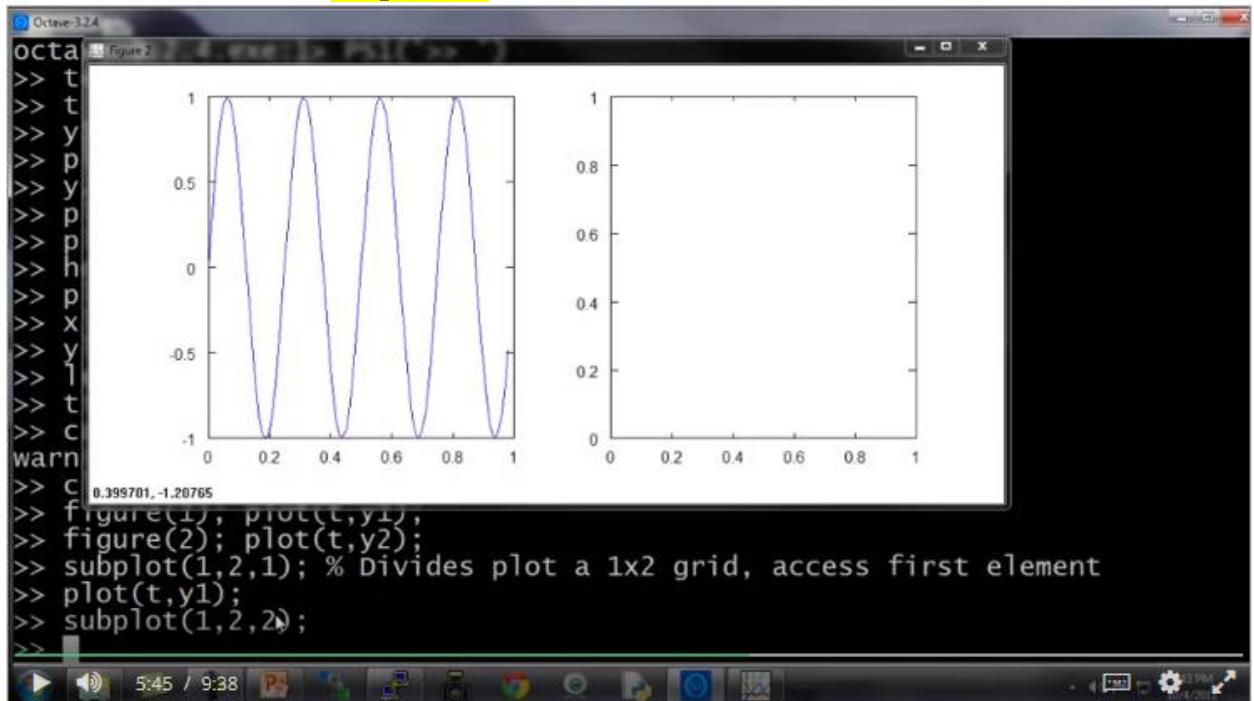
Here's one other neat command that I often use, which is the **subplot command**. So, we're going to use **subplot 1, 2, 1**. What it does it sub-divides the plot into a one-by-two grid with the first 2 parameters are, and it starts to access the first element. That's what the final parameter 1 is, right?

```

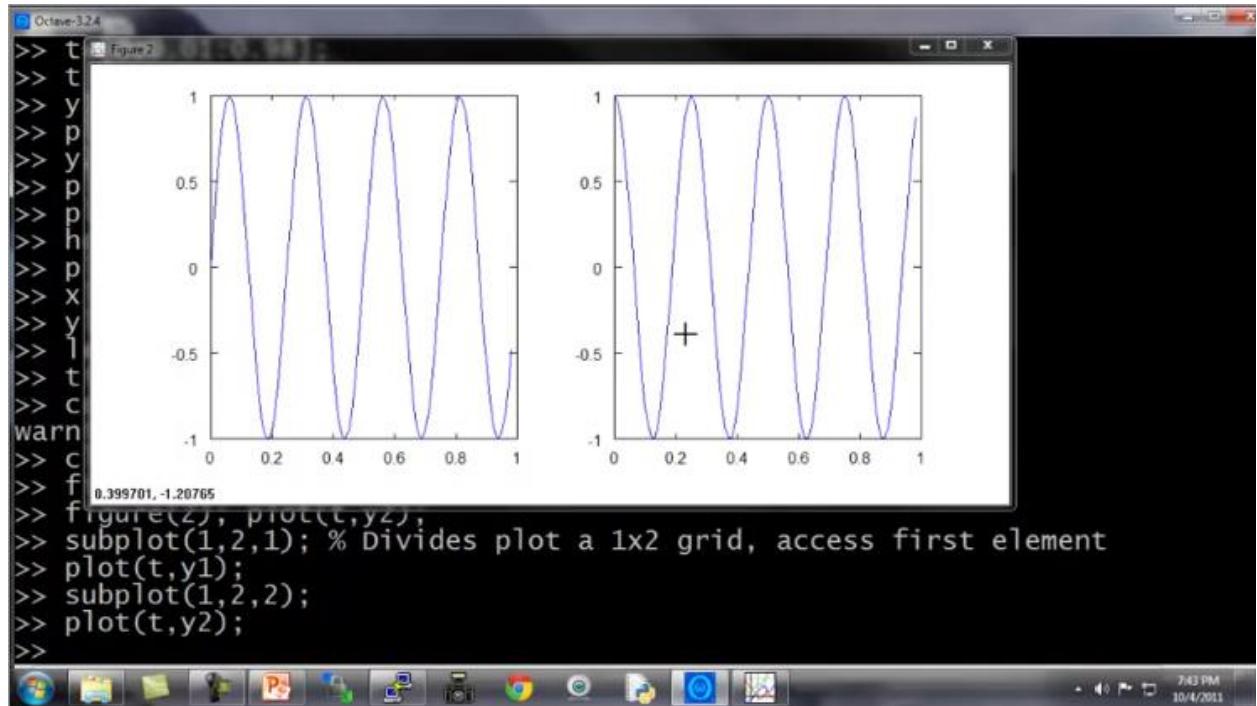
octa
>> t
>> t
>> y
>> p
>> p
>> h
>> p
>> x
>> y
>> l
>> t
>> c
warn
>> c
0.519985, 1.08924
>> figure(1); plot(t,y1);
>> figure(2); plot(t,y2);
>> subplot(1,2,1); % Divides plot a 1x2 grid, access first element
>>

```

So, **divide my figure into a one by two grid**, and I want to access the first element right now. And so, if I type that in, this product, this figure, is on the left. And if I **plot t, y1**, it now fills up this first element. And if I'll do **subplot 122**.

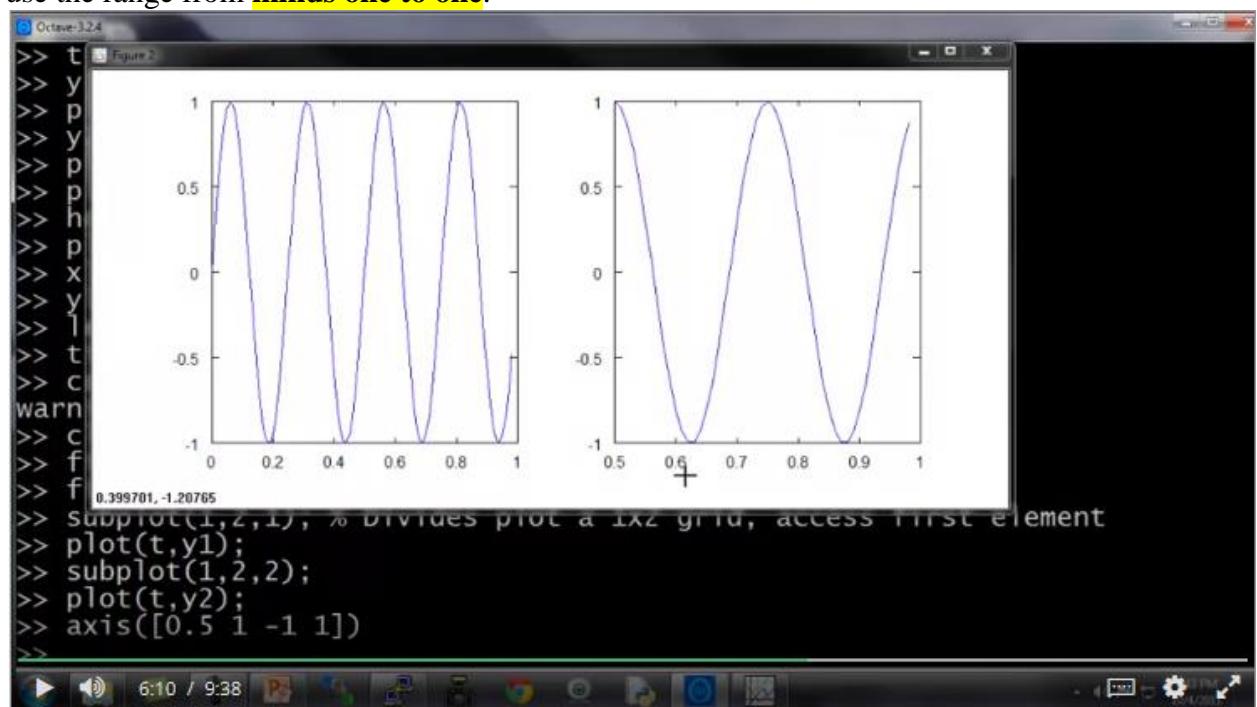


I'm going to start to access the second element and **plot t, y2** will throw in **y2** in the right hand side, or in the second element.



### axis command – To Change Scale

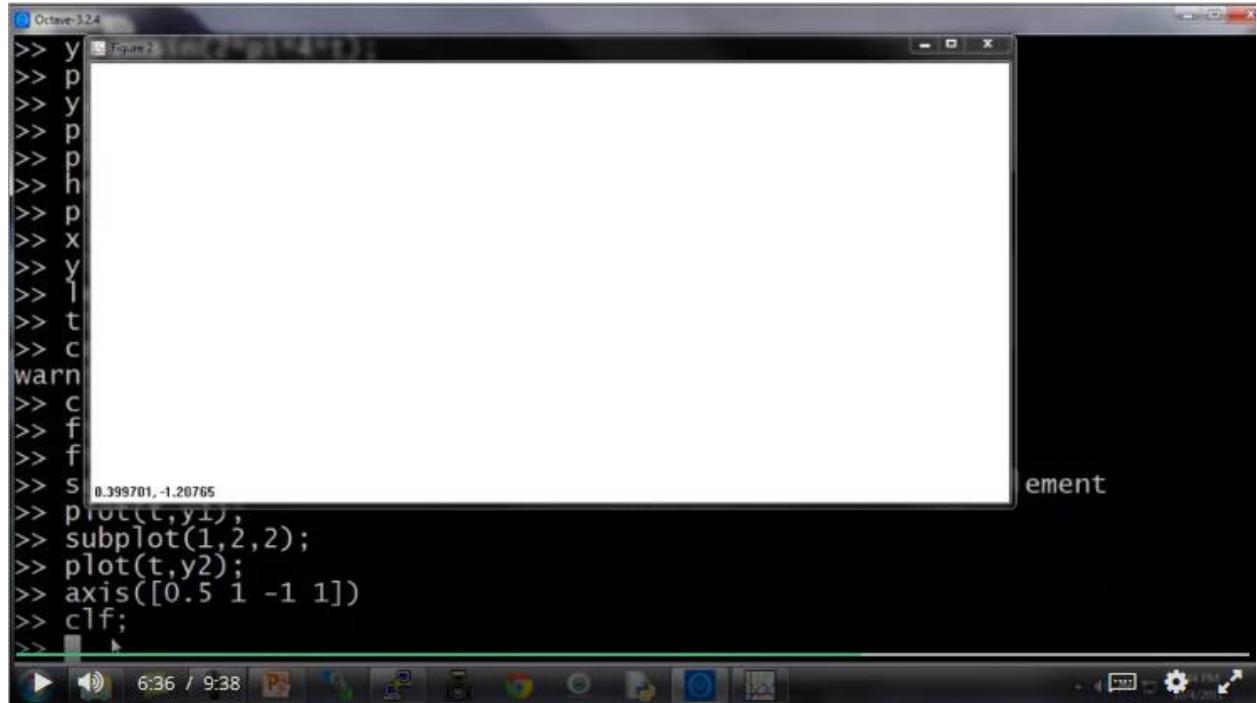
And last command, you can also change the axis scales and change axis these to 1, .5, 1 minus 1 to 1 and this sets the x range and y range for the figure on the right, and concretely, it assess the **horizontal major values** in the figure on the right to make sure **0.5 to 1**, and the **vertical axis values** use the range from **minus one to one**.



And, you know, you don't need to memorize all these commands. If you ever need to change the axis or you need to know is that, you know, there's an axis command and you can already get the details from the usual octave **help command**.

### clf command – To Clear Figure

Finally, just a couple last commands CLF clear is a figure and here's one unique trait.

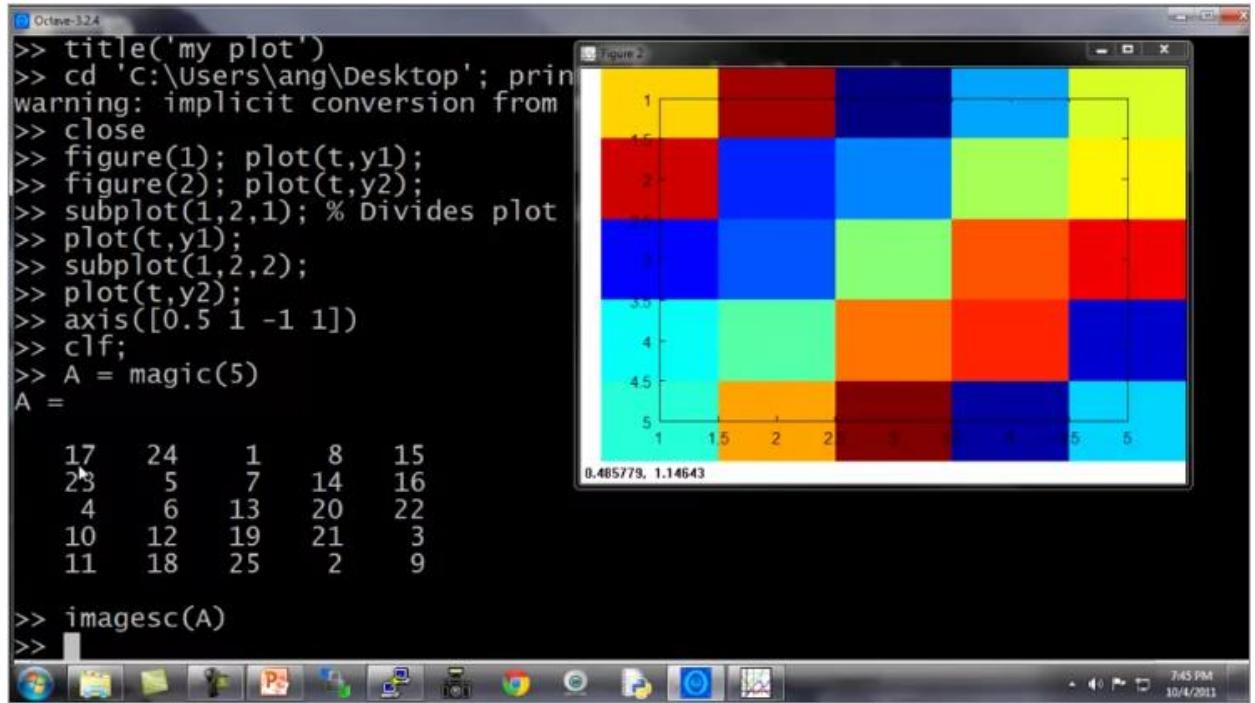


The screenshot shows the Octave 3.2.4 interface. The command window contains the following code:

```
>> y
>> p
>> y
>> p
>> ph
>> p
>> x
>> y
>> l
>> t
>> c
>> warn
>> c
>> f
>> f
>> S
>> plot(t,y1);
>> subplot(1,2,2);
>> plot(t,y2);
>> axis([0.5 1 -1 1])
>> clf;
>>
```

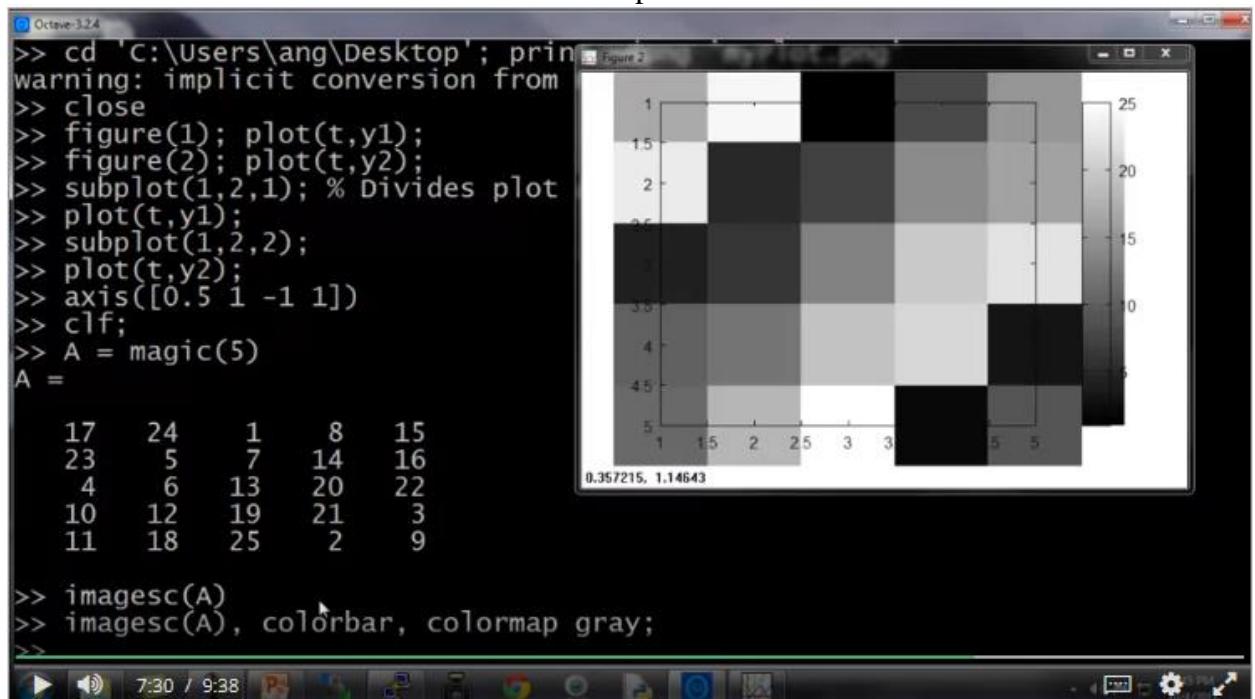
The plot window titled "Figure 2" is visible in the background, showing a plot with two subplots. The status bar at the bottom indicates the time as 6:36 / 9:38.

Let's set A to be equal to a 5 by 5 magic squares a. So, A is now this 5 by 5 matrix, does a neat trick that I sometimes use to visualize the matrix, which is I can use **imagesc** of what this will do is plot a five by five matrix, a five by five grid of color where the different colors correspond to the different values in the A matrix.



So concretely, I can also do color bar. Let me use a more sophisticated command,

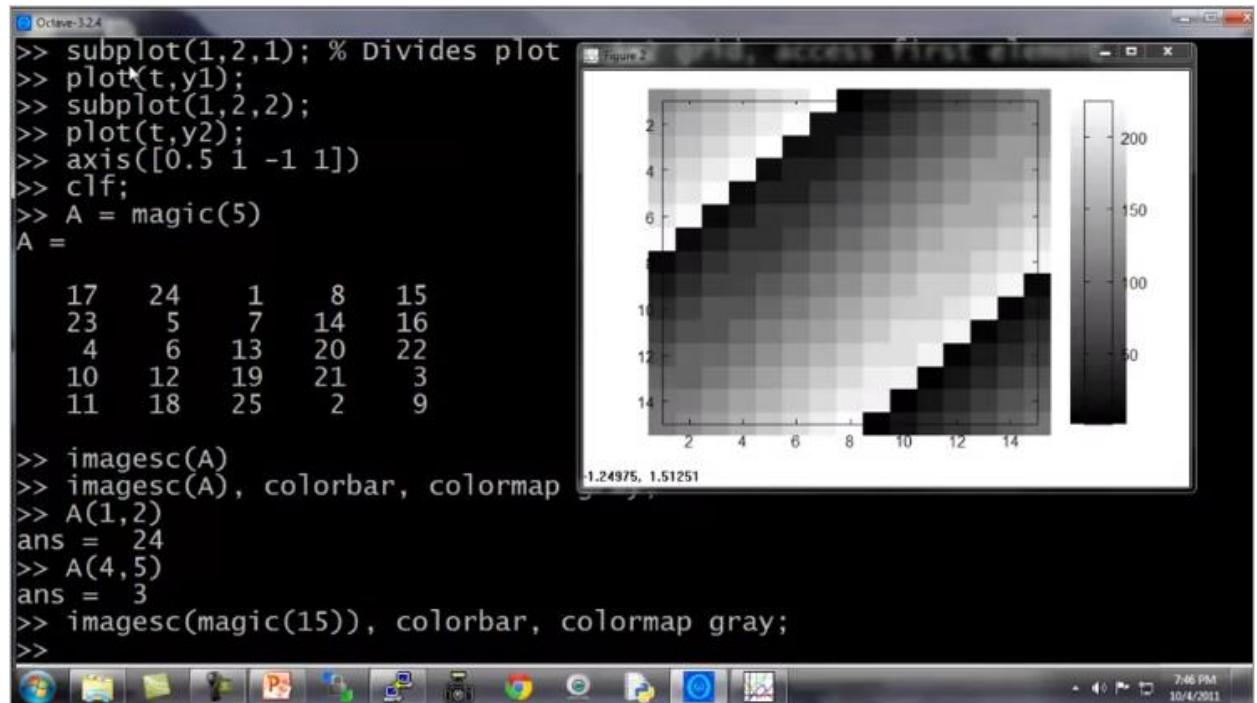
**imagesc A colormap gray**. This is actually running three commands at a time. I'm running imagesc then running colorbar, then running colormap gray. And what this does, is it sets a color map, so a gray color map, and on the right it also puts in this colorbar. And so this color bar shows what the different shades of color correspond to.



Concretely, the upper left element of the A matrix is 17, and so that corresponds to kind of a mint shade of gray. Whereas in contrast the second element of A--sort of the 1 2 element of A--is

24. Right, so it's A 1 2 is 24. So that corresponds to this square out here, which is nearly a shade of white. And the small value, say A--what is that? A 4 5, you know, is a value 3 over here that corresponds-- you can see on my color bar that it corresponds to a much darker shade in this image.

So here's another example, I can plot a larger, you know, here's a magic 15 that gives you a 15 by 15 magic square and this gives me a plot of what my 15 by 15 magic squares values looks like.



### Comma Chaining of Commands

And finally to wrap up this video, what you've seen me do here is use comma chaining of function calls. Here's how you actually do this. If I type A equals 1, B equals 2, C equals 3, and hit Enter, then this is actually carrying out three commands at the same time. Or really carrying out three commands, one after another, and it prints out all three results.

```

>> a=1, b=2, c=3
a = 1
b = 2
c = 3

```

And this is a lot like A equals 1, B equals 2, C equals 3, except that if I use semicolons instead of a comma, it doesn't print out anything.

```

>> a=1; b=2; c=3;
>>

```

So, this, you know, this thing here we call comma chaining of commands, or comma chaining of function calls. And, it's just another convenient way in Octave to put multiple commands like **imagesc** **colorbar**, **colonmap** to put multi-commands on the same line.

So, that's it. You now know how to plot different figures and octave, and in next video the next main piece that I want to tell you about is **how to write control statements** like if, while, for statements and octave as well as hard to define and use functions

## Control Statements: for, while, if statement

In this video, I'd like to tell you how to write control statements for your Octave programs, so things like "for", "while" and "if" statements and also how to define and use functions.

Here's my Octave window.

```
octave-3.2.4.exe:1> PS1('>> ')
>>
```

### for loop

Let me first show you how to use a "**for**" loop. I'm going to start by setting v to be a **10 by 1 vector** of 0. Now, here's I write a "for" loop for I equals 1 to 10. That's for I equals Y colon 10. And let's see, I'm going to set V of I equals two to the power of I, and finally end. The white space does not matter, so I am putting the spaces just to make it look nicely indented, but you know spacing doesn't matter. But if I do this, then the result is that V gets set to, you know, two to the power one, two to the power two, and so on. So this is syntax for I equals one colon 10 that makes I loop through the values one through 10.

<pre>&gt;&gt; v=zeros(10,1) v = 0 0 0 0 0 0 0 0 0 0</pre>	<pre>&gt;&gt; v v = 2 4 8 16 32 64 128 256 512 1024</pre>
	<pre>&gt;&gt; for i=1:10, &gt; v(i) = 2^i; &gt; end;</pre>

And by the way, you can also do this by setting your indices equals one to 10, and so the indices in the array from one to 10.

<pre>&gt;&gt; indices=1:10; &gt;&gt; indices indices = 1    2    3    4    5    6    7    8    9    10</pre>
--

You can also write for I equals indices. And this is actually the same as if I equals one to 10. You can do, you know, display I and this would do the same thing.

```
>> for i=indices,
>     disp(i);
> end;
1
2
3
4
5
6
7
8
9
10
```

### while loop

So, that is a "for" loop, if you are familiar with "break" and "continue", there's "break" and "continue" statements, you can also use those inside loops in octave, but first let me show you how a **while loop** works. So, here's my vector V. Let's write the while loop. I equals 1, while I is less than or equal to 5, let's set V I equals one hundred and increment I by one, end. So this says what? I starts off equal to one and then I'm going to set V I equals one hundred and increment I by one until I is, you know, greater than five. And as a result of that, whereas previously V was this powers of two vector. I've now taken the first five elements of my vector and overwritten them with this value one hundred. So that's a syntax for a while loop.

```
>> v
v =
100
100
100
100
100
64
128
256
512
1024
```

```
>> i = 1;
>> while i <= 5,
>     v(i) = 100;
>     i = i+1;
> end;
```

### break and if statement

Let's do another example. I equals one, while true, and here I wanted to show you how to use a **break statement**. Let's say V I equals 999 and I equals i+1 if i equals 6 break and end. And this is also our first use of **if statement**, so I hope the logic of this makes sense. Since I equals one and, you know, increment loop. While repeatedly increment i by 1, and then when i gets up to 6, **do a break** which breaks here although the while do and so, the effective is should be to take the first five elements of this vector V and set them to 999. And yes, indeed, we're taking V and overwritten the first five elements with 999. So, this is the syntax for "if" statements, and for

"while" statement, and notice the end. We have two ends here. This end here ends the if statement and the second end here ends the while statement.

```
>> v
v =

999
999
999
999
999
64
128
256
512
1024

>> i=1;
>> while true,
>     v(i) = 999;
>     i = i+1;
>     if i == 6,
>         break;
>     end;
> end;
```

### if-else statement

Now let me show you the more general syntax for how to use an if-else statement. So, let's see, V 1 is equal to 999, let's type V1 equals to 2 for this example. So, let me type if V 1 equals 1 display the value as one. Here's how you write an else statement, or rather here's an else if: V 1 equals 2. This is, if in case that's true in our example, display the value as 2, else display, the value is not one or two. Okay, so that's a if-else if-else statement it ends. And of course, here we've just set v 1 equals 2, so hopefully, yup, displays that the value is 2.

```
>> v(1)
ans = 999
>> v(1) = 2;
>> if v(1)==1,
>     disp('The value is one');
> elseif v(1) == 2,
>     disp('The value is two');
> else
>     disp('The value is not one or two.');
> end;
The value is two
>>
```

### exit and quit commands

And finally, I don't think I talked about this earlier, but if you ever need to exit Octave, you can type the exit command and you hit enter that will cause Octave to quit or the 'q'--quits command also works.

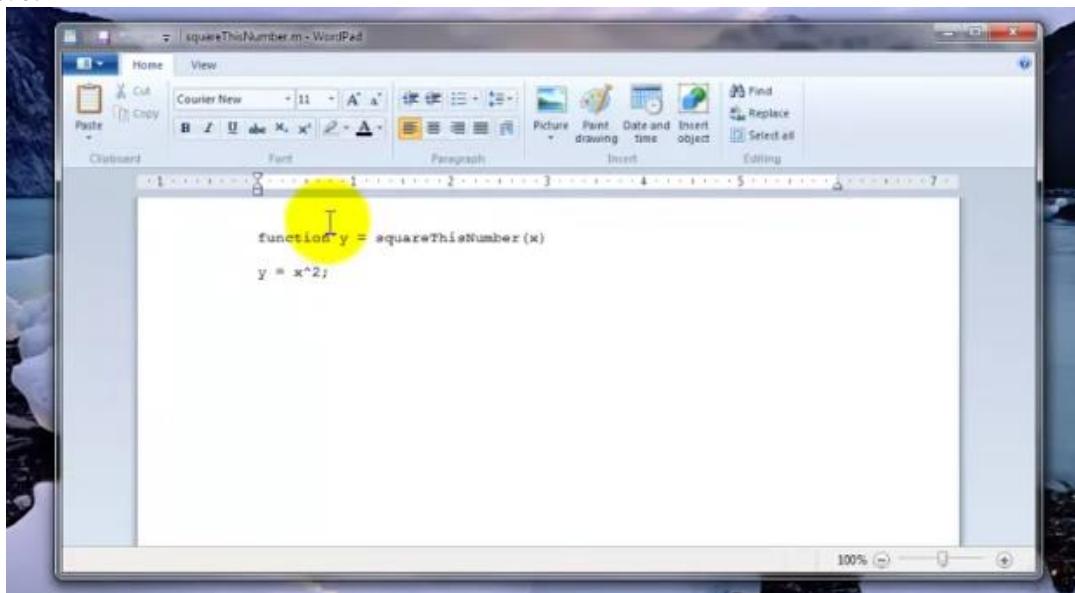
```
>> exit
>> quit
```

### functions

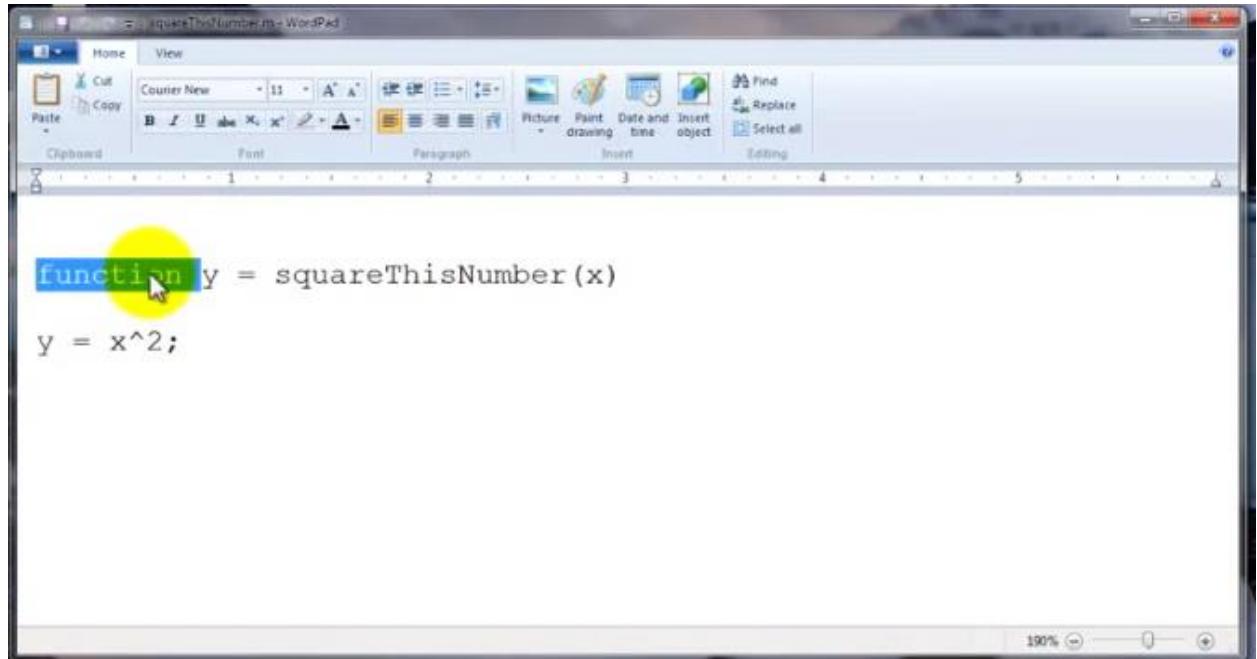
Finally, let's talk about functions and how to define them and how to use them. Here's my desktop, and I have predefined a file or pre-saved on my desktop a file called "squarethisnumber.m". This is how you define functions in Octave. You create a file called, you know, with your function name and then ending in .m, and when Octave finds this file, it knows that this where it should look for the definition of the function "squarethisnumber.m".



Let's open up this file. Notice that I'm using the **Microsoft program Wordpad** to open up this file. I just want to encourage you, if you're using Microsoft Windows, to use Wordpad rather than Notepad to open up these files, if you have a different text editor that's fine too, but **notepad sometimes messes up the spacing**. If you only have Notepad, that should work too, that could work too, but if you have Wordpad as well, I would rather use that or some other text editor, if you have a different text editor for editing your functions. So, here's how you define the function in Octave.



Let me just zoom in a little bit. And this file has just three lines in it. The first line says function Y equals square root number of X, this tells Octave that I'm gonna return the value Y, I'm gonna return one value and that the value is going to be saved in the variable Y and moreover, it tells Octave that this function has one argument, which is the argument X, and the way the function body is defined, if Y equals X squared.



So, let's try to call this function "square", this number 5, and this actually isn't going to work, and Octave says square this number it's undefined. That's because Octave doesn't know where to find this file.

```
>> squareThisNumber(5)
error: `squareThisNumber' undefined near line 18 column 1
```

So as usual, let's use PWD, or not in my directory, so let's see this c:\users\ang\Desktop. That's where my desktop is. Oops, a little typo there. Users ANG desktop and if I now type square root number 5, it returns the answer 25.

```
>> pwd
ans = C:\Octave\3.2.4_gcc-4.4.0\bin
>> cd 'C:\User\ang\Desktop'
error: C:\User\ang\Desktop: No such file or directory
>> cd 'C:\Users\ang\Desktop'
>> squareThisNumber(5)
ans = 25
>>
```

## Search Path

As kind of an advanced feature, this is only for those of you that know what the term search path means. If you want to modify the Octave search path and you could, you just think of this next part as advanced or optional material. Only for those who are either familiar with the concepts of search paths and programming languages, but you can use the term **addpath**, C colon, slash users/ANG/Desktop to add that directory to the Octave search path so that even if you know, go to some other directory it can still, Octave still knows to look in the users ANG desktop directory for functions so that even though I'm in a different directory now, it still knows where to find the square this number function. Okay?

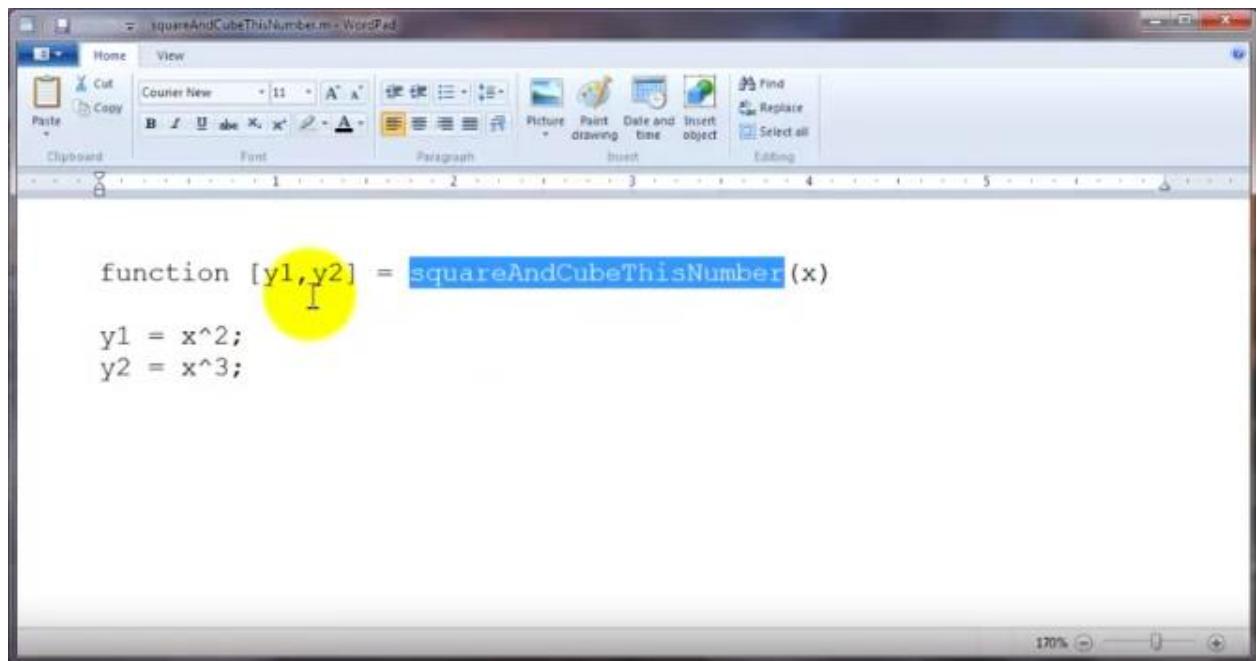
```
>> % Octave search path (advanced/optional)
>> addpath('C:\Users\ang\Desktop')
>> cd 'C:\'
>> squareThisNumber(5)
ans = 25
```

### cd command

But if you're not familiar with the concept of search path, don't worry about it. Just make sure as you use the CD command to go to the directory of your function before you run it and that actually works just fine.

### Multiple Return Values

One concept that Octave has that **many other programming languages don't** is that it can also let you define functions that **return multiple values** or multiple arguments. So here's an example of that. Define the function called square and cube this number X and what this says is this function returns 2 values, y1 and y2. This follows, y1 is x squared, y2 is x cubed. And what this does is this really returns 2 numbers. So, some of you depending on what programming language you use, if you're familiar with, you know, C/C++ what they offer. Often, we think of the function as returning just one value. But just so the syntax in Octave that should return multiple values.

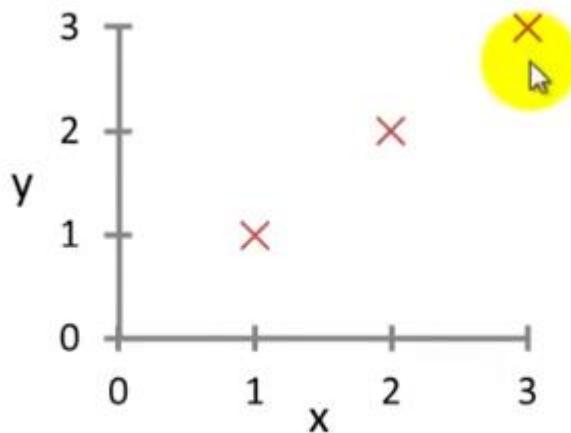


Now back in the Octave window. If I type, you know, a, b equals square and cube this number 5 then a is now equal to 25 and b is equal to the cube of 5 equal to 125. So, this is often convenient if you needed to define a function that returns multiple values.

```
>> [a,b] = squareAndCubeThisNumber(5);
>> a
a = 25
>> b
b = 125
>>
```

### Compute Cost Function $J$ of $\theta$

Finally, I'm going to show you just one more sophisticated example of a function. Let's say I have a data set that looks like this, with data points at 1, 1, 2, 2, 3, 3. And what I'd like to do is to **define an octave function to compute the cost function  $J$  of theta** for different values of theta.



**Goal:** Define a function to compute the cost function  $J(\theta)$ .

First let's put the data into octave. So I set my design matrix to be 1,1; 1,2;1,3. So, this is my design matrix  $x$  with  $x_0$  the first column being the said term and the second term being you know, my the x-values of my three training examples. And let me set  $y$  to be 1-2-3 as follows, which were the y axis values.

<pre>&gt;&gt; x = [1 1; 1 2; 1 3] x = 1 1 1 2 1 3</pre>	<pre>&gt;&gt; y = [1; 2; 3] y = 1 2 3</pre>
---	---

So let's say theta is equal to 0 semicolon 1.

```

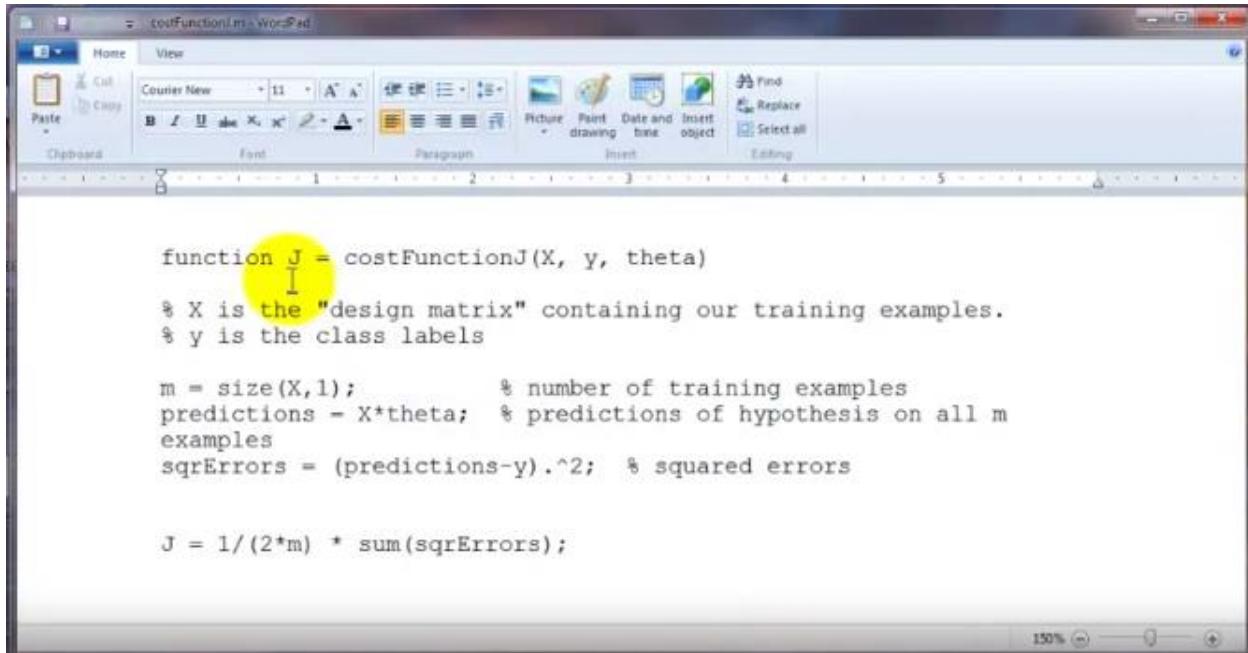
>> X = [1 1; 1 2; 1 3]
X =
    1   1
    1   2
    1   3

>> y = [1; 2; 3]
y =
    1
    2
    3

>> theta = [0;1];

```

Here at my desktop, I've predefined does cost function j and if I bring up the definition of that function it looks as follows.



The screenshot shows a Microsoft Word document window titled "costFunctionJ.m - Microsoft Word". The document contains the following MATLAB code:

```

function J = costFunctionJ(X, y, theta)
% X is the "design matrix" containing our training examples.
% y is the class labels

m = size(X,1); % number of training examples
predictions = X*theta; % predictions of hypothesis on all m
examples
sqrErrors = (predictions-y).^2; % squared errors

J = 1/(2*m) * sum(sqrErrors);

```

So function j, equals cost function j equals x y theta, some comments, specifying the inputs and then vary few steps set m to be the number trading examples, thus the number of rows in x. Compute the predictions, predictions equals x times theta, and this is a comment that's wrapped around, so this is probably the preceding comment line. Computing square errors by taking the difference between your predictions and the y values and taking the element of y squaring and then finally computing the cost function J. And **Octave knows that J is a value that I want to return because J appeared here in the function definition.**

Feel free by the way to pause this video if you want to look at this function definition for longer and kind of make sure that you understand the different steps.

But when I run it in Octave, I run j equals cost function j x y theta. It computes. Oops, made a typo there. It should have been capital X. It computes J equals 0 because if my data set was, you know, 123, 123 then setting, theta 0 equals 0, theta 1 equals 1, this gives me exactly the 45-degree line that fits my data set perfectly.

```
>> j = costFunctionJ(x,y,theta)
error: `x' undefined near line 31 column 19
error: evaluating argument list element number 1
error: evaluating argument list element number 1
>> j = costFunctionJ(X,y,theta)
j = 0
>> █
```

Whereas in contrast if I set theta equals say 0, 0, then this hypothesis is predicting zeroes on everything the same, theta 0 equals 0, theta 1 equals 0 and I compute the cost function then it's 2.333

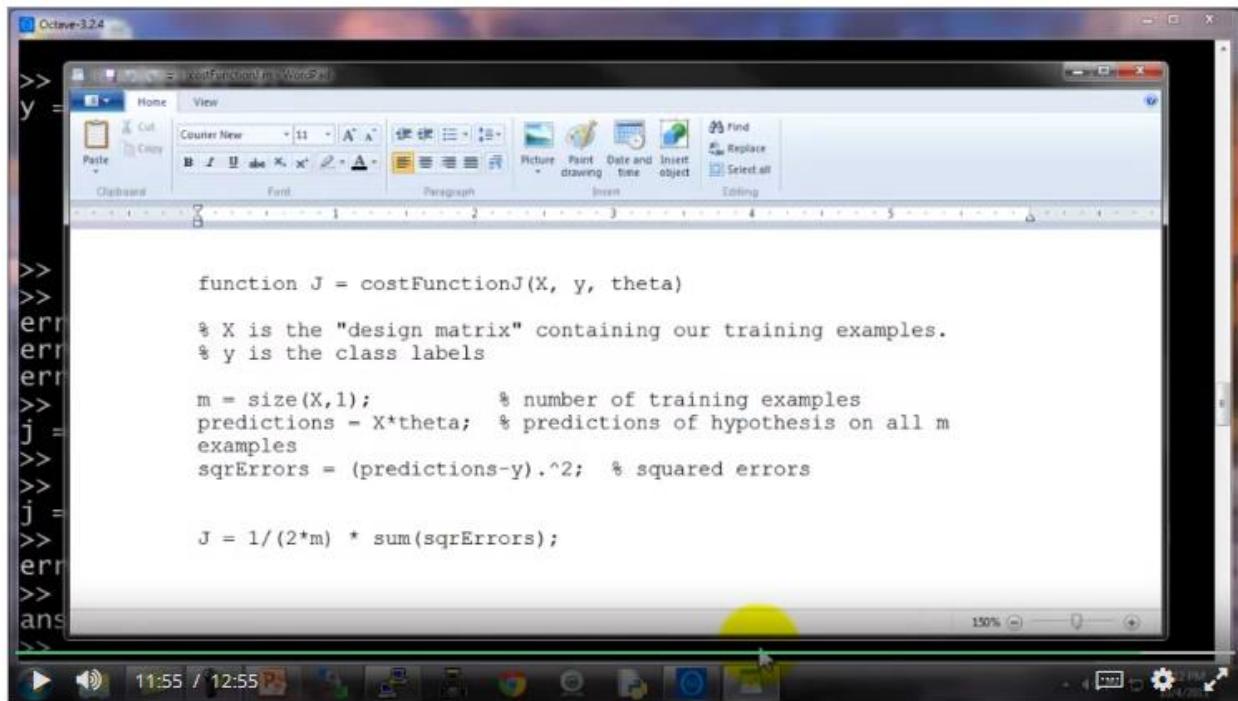
```
>> theta = [0;0];
>> j = costFunctionJ(X,y,theta)
j = 2.3333
>> █
```

And that's actually equal to 1 squared, which is my **squared error** on the first example, plus 2 squared, plus 3 squared and then divided by 2m, which is **2 times number of training examples**, which is indeed 2.33

```
>> (1^2 + 2^2 + 3^2) / (2*3)
ans = 2.3333
```

And so, that sanity checks that this function here is, you know, computing the correct cost function and these are the couple examples we tried out on our simple training example. And so that sanity tracks that the cost function J, as defined here, that it is indeed, you know, seeming to compute the correct cost function, at least on our simple training set that we had here with X and

Y being this simple training example that we solved.



```

>> y =
>>
>> function J = costFunctionJ(X, y, theta)
>> % X is the "design matrix" containing our training examples.
>> % y is the class labels
>> m = size(X,1); % number of training examples
>> predictions = X*theta; % predictions of hypothesis on all m
>> examples
>> sqrErrors = (predictions-y).^2; % squared errors
>> J = 1/(2*m) * sum(sqrErrors);
>>
>> ans
>>

```

So, now you know how to right control statements like for loops, while loops and if statements in octave as well as how to define and use functions.

In the next video, I'm going to just very quickly step you through the **logistics of working on and submitting problem sets for this class** and **how to use our submission system**. And finally, after that, in the final octave tutorial video, I wanna tell you about **vectorization**, which is an idea for how to make your octave programs run much fast.

## Vectorization

In this video I like to tell you about the idea of **Vectorization**.

So, whether you using Octave or a similar language like MATLAB or whether you're using Python NumPy, R, Java, C, C++, all of these languages have either built into them or have regularly and easily accessible different **numerical linear algebra libraries**. They're usually very well written, highly optimized, often sort of developed by people that have PhDs in numerical computing or they're really specialized in numerical computing.

And when you're implementing machine learning algorithms, if you're able to **take advantage of these linear algebra libraries** or these numerical linear algebra libraries, and make some routine calls to them rather than sort of write code yourself to do things that these libraries could be doing. If you do that, then often you **get code that, first, is more efficient**, so you just **run more quickly** and take better advantage of any parallel hardware your computer may have and so on.

And second, it also means that you end up with less code that you need to write, so it's **a simpler implementation** that is therefore maybe also more likely to be bug free.

And as a concrete example, rather than writing code yourself to multiply matrices, if you let Octave do it by typing A times B, that would use a very efficient routine to multiply the two matrices. And there's a bunch of examples like these, where if you use appropriate vectorized implementations you get much simpler code and much more efficient code.

### Let's look at some examples.

Here's our **usual hypothesis for linear regression**, and if you want to compute  $h(x)$ , notice that there's a sum on the right. And so one thing you could do is, compute the sum from  $j = 0$  to  $j = n$  yourself. Another way to think of this is to **think of  $h(x)$  as theta transpose x**, and what you can do is, think of this as you are computing this **inner product between two vectors** where **theta is your vector, say,  $\theta_0, \theta_1, \theta_2$** . If **you have two features**, if  $n$  equals two, and if you **think x as this vector,  $x_0, x_1, x_2$** , and these two views can give you two different implementations.

### Vectorization example.

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

$$= \theta^T x$$

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix}$$

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

theta (1)  
theta (2)  
theta (3)

### Unvectorized implementation

```

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

```

### Vectorized implementation

```

→ prediction = theta' * x;

```

Here's an **unvectorized implementation** for how to compute  $h(x)$ , and by unvectorize I mean without vectorization.

We might first initialize prediction just to be 0.0. This is going to eventually be your prediction, is eventually going to be  $h(x)$ , and then I'm going to have a for loop for  $j=1$  through  $n+1$ , prediction gets incremented by  $\theta(j) * x(j)$ . So it's kind of this expression over here. By the way, I should mention, in these vectors that I wrote over here, I had these vectors being 0 index. So I had theta 0, theta 1, theta 2. But because **MATLAB is one index**, theta 0 in that MATLAB, we would end up representing as theta 1 and the second element ends up as theta 2 and this third element may end up as theta 3, just because our **vectors in MATLAB are indexed starting**

**from 1**, even though I wrote theta and x here, starting indexing from 0, which is why here I have a for loop. j goes from 1 through n+1 rather than j goes through 0 up to n, right? But so this is an unvectorized implementation in that we have a for loop that is summing up the n elements of the sum.

In contrast, **here's** how you would write a **vectorized implementation**, which is that you would **think of x and θ as vectors**.

You just said prediction =  $\theta^T * x$ .

You're just computing like so. So instead of writing all these lines of code with a for loop, you instead just have one line of code. And **what this line of code on the right will do is, it will use Octaves highly optimized numerical linear algebra routines to compute this inner product between the two vectors, θ and X**, and not only is the vectorized implementation simpler, it will also run much more efficiently.

So that was octave, but the issue of vectorization applies to other programming language as well. **Let's look on the example in C++.** Here's what an **unvectorized implementation** might look like.

#### Unvectorized implementation

```
→ double prediction = 0.0;
→ for (int j = 0; j <= n; j++)
    prediction += theta[j] * x[j];
```

We again initialize prediction to 0.0 and then we now have a for loop for  $j = 0$  up to  $n$ .

Prediction  $\leftarrow \theta_j * x[j]$ , where again, you have this explicit for loop that you write yourself.

In contrast, using a good numerical linear algebra library in C++, you could write a function like, or rather, in contrast, **using a good numerical linear algebra library in C++, you can instead write code that might look like this.**

#### Vectorized implementation

```
double prediction
= theta.transpose() * x;
```

So depending on the details of your numerical linear algebra library, you might be able to have an object, this is a **C++ object**, which is **vector θ**, and a **C++ object** which is **vector x**, and you just take  $\theta.\text{transpose} * x$ , where this **times** becomes a **C++ sort of overload operator** so you can just multiply these two vectors in C++.

And depending on the details of your numerical linear algebra library, you might end up using a slightly different syntax, but by relying on the library to do this inner product, you can get a much simpler piece of code and a much more efficient one.

#### **Let's now look at a more sophisticated example.**

Just to remind you, **here's our update rule for a gradient descent of a linear regression**. And so we update  $\theta_j$  using this rule for all values of  $j = 0, 1, 2$ , and so on.

#### **Gradient descent**

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

*j = 0, 1, 2*

And if I just write out these equations for theta 0, theta 1, theta 2, assuming we have two features, so  $n = 2$ . Then these are the updates we perform for theta 0, theta 1, theta 2, where you might remember my saying in an earlier video, that these should be simultaneous updates.

$$\begin{aligned} \theta_0 &:= \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)} \\ \theta_1 &:= \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_1^{(i)} \\ \theta_2 &:= \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_2^{(i)} \end{aligned}$$

Simultaneous updates.

So, let's see if we can come up with a vectorizing implementation of this. Here are my same three equations written in a slightly smaller font, and you can imagine that one way to implement these three lines of code is to have a for loop that says for  $j = 0, 1$  through 2 to update  $\theta_j$ , or something like that. But instead, let's come up with a vectorized implementation and see if we can have a simpler way to basically compress these three lines of code or a for loop that effectively does these three steps one set at a time. Let's see if we can take these three steps and compress them into one line of vectorized code. Here's the idea.

$$\begin{aligned} \theta_0 &:= \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)} \\ \theta_1 &:= \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_1^{(i)} \\ \theta_2 &:= \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_2^{(i)} \end{aligned} \quad (n=2)$$

Vectorized implementation:

$$\theta := \theta - \alpha \delta$$

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

What I'm going to do is, **I'm going to think of  $\theta$  as a vector**, and I'm gonna **update  $\theta$  as  $\theta$  minus alpha times some other vector delta**, where delta's is going to be equal to 1 over m, sum from  $i = 1$  through m. And then this term over on the right, okay? So, let me explain what's going on here. **Here, I'm going to treat theta as a vector**, so this is n plus one dimensional vector, and I'm saying that theta gets here updated as that's a vector,  $\mathbf{R}^{n+1}$ . Alpha is a real number, and **delta, here is a vector**. So, this subtraction operation, that's a **vector subtraction**, okay? Cuz alpha times delta is a vector, and so I'm saying theta gets this vector, alpha times delta subtracted from it.

**So, what is a vector delta?** Well this vector delta, looks like this, and what it's meant to be is really meant to be this thing over here. Concretely, **delta will be a n plus one dimensional vector**, and the very first element of the vector delta is going to be equal to that. So, if we have the delta, if we index it from 0, if it's delta 0, delta 1, delta 2, what I want is that delta 0 is equal to this first box in green up above.

And indeed, you might be able to convince yourself that delta 0 is this  **$1/m (\text{sum of } (\mathbf{h}(\mathbf{x}^i) \text{ minus } \mathbf{y}^{(i)}) \text{ times } \mathbf{x}^{(i)}_0)$** . So, let's just make sure that we're on this same page about how delta really is computed. Delta is 1 over m times this sum over here, and you know what is this sum? Well, **this term over here, that's a real number**, and the second term over here,  **$\mathbf{x}^{(i)}$ , this term over there is a vector**, right, because  **$\mathbf{x}^{(i)}$** , may be a vector that would be, say,  **$\mathbf{x}^{(i)}_0, \mathbf{x}^{(i)}_1, \mathbf{x}^{(i)}_2$** , right, and what is the summation? Well, what the summation is saying is that, this term, that is this term over here, this is equal to,  **$(\mathbf{h}(\mathbf{x}^{(1)}) - \mathbf{y}^{(1)}) * \mathbf{x}^{(1)} + (\mathbf{h}(\mathbf{x}^{(2)}) - \mathbf{y}^{(2)}) * \mathbf{x}^{(2)} + \dots$** , okay?

Because this is summation of i, so as i ranges from i = 1 through m, you get these different terms, and you're summing up these terms here. And the meaning of these terms, this is a lot like if you remember actually from the earlier quiz in this, right, you saw this equation. We said that in order to vectorize this code we will instead said  $u(j) = 2v(j) + 5w(j)$ . So we're saying that the **vector u is equal to two times the vector v plus five times the vector w**. So this is an example of how to add different vectors and this summation's the same thing. This is saying that the summation over here is just some real number, right? That's kinda like the number two or some other number times the vector, x1. So it's kinda like 2v or say some other number times x1, and then plus instead of 5w we instead have some other real number, plus some other vector, and then you add on other vectors, plus dot, dot, dot, plus the other vectors, which is why, over all, this thing over here, that whole quantity, that **delta is just some vector**.

And concretely, the three elements of delta correspond to, if n = 2, the three elements of delta correspond exactly to this thing, to the second thing, and this third thing. Which is why when you update theta according to theta minus alpha delta, we end up carrying exactly the same simultaneous updates as the update rules that we have up top.

So, I know that there was a lot that happened on this slide, but again, feel free to pause the video and if you aren't sure what just happened I'd encourage you to step through this slide to make sure you understand why is it that this update here with this definition of delta, right, why is it that that's equal to this update on top? And if it's still not clear, one insight is that, this thing over here, that's exactly the vector x, and so we're just taking all three of these computations, and compressing them into one step with this vector delta, which is why we can come up with a vectorized implementation of this step of linear regression this way.

So, I hope this step makes sense and do look at the video and make sure you can understand it.

In case you don't understand quite the equivalence of this map, if you implement this, this turns out to be the right answer anyway. So, even if you didn't quite understand equivalence, if you just implement it this way, you'll be able to get linear regression to work. But if you're able to figure out why these two steps are equivalent, then hopefully that will give you a better understanding of vectorization as well.

And finally, if you are implementing linear regression using more than one or two features, so sometimes we use linear regression with 10's or 100's or 1,000's of features. But if you use the vectorized implementation of linear regression, you'll see that will run much faster than if you had, say, your old for loop that was updating theta zero, then theta one, then theta two yourself. So, using a vectorized implementation, you should be able to get a much more efficient implementation of linear regression. And when you vectorize later algorithms that we'll see in this class, there's good trick, whether in Octave or some other language like C++, Java, for getting your code to run more efficiently.

## Octave Installation Suggestions from Mentor

Note (added April 19, 2018) - Updated Nov. 1, 2019

<https://www.coursera.org/learn/machine-learning/discussions/all/threads/vgCyrQoMEeWv5yIAC00Eog?page=2>

---

The course programming assignments were originally developed for Octave version 3.2.4.

The course was later updated and tested with MATLAB R2015a and Octave version 3.8.2.

Octave version 3.2.4 and version 3.8.2 are not recommended - especially if you use linux or have a Mac computer.

You **cannot use Octave 4.0.0**. It has a fatal error when you run the "submit" script. Use the instructions in the Resources menu

"Installation Issues" or "Tips on Octave OS X" to get a newer version.

Octave version 5.1.0 also has a defect - avoid it.

---

The **course presently supports** two different and incompatible versions of the programming assignment zip files:

**Octave version 4.4.1** and older versions of desktop MATLAB.

MATLAB Online or desktop MATLAB 2019b.

See Week 2 for instructions on setting up your programming environment.

---

## Important Notes for New ML Students

<https://www.coursera.org/learn/machine-learning/discussions/all/threads/v2YppY8FEEeWleBJxvl1elQ>

# Installing Octave on Windows

## Installing Octave on Windows

Use this link to install Octave for windows:

[http://wiki.octave.org/Octave\\_for\\_Microsoft\\_Windows](http://wiki.octave.org/Octave_for_Microsoft_Windows)

Octave on Windows can be used to submit programming assignments in this course but will likely need a patch provided in the discussion forum. Refer to <https://www.coursera.org/learn/machine-learning/discussions/vgCyrQoMEeWv5yIAC00Eog?> for more information about the patch for your version.

"Warning: Do not install Octave 4.0.0"; checkout the "Resources" menu's section of "Installation Issues".

# Logistic Regression

## Classification and Representation

### Classification

In this and the next few videos, I want to start to talk about **classification problems**, where the **variable y** that you want to predict is **discrete valued**. We'll develop an algorithm called **logistic regression**, which is one of the most popular and most widely used learning algorithms today.

Here are some examples of classification problems.

### Classification

- Email: Spam / Not Spam?
- Online Transactions: Fraudulent (Yes / No)?
- Tumor: Malignant / Benign ?

Earlier we talked about email spam classification as an example of a classification problem. Another example would be classifying online transactions. So if you have a website that sells stuff and if you want to know if a particular transaction is fraudulent or not, whether someone is using a stolen credit card or has stolen the user's password. There's another classification problem. And earlier we also talked about the example of classifying tumors as cancerous, malignant or as benign tumors. In all of these problems the variable that we're trying to predict is a variable **y** that we can think of as taking on two values either zero or one, either spam or not spam, fraudulent or not fraudulent, related malignant or benign. Another name for the class that we denote with **zero is the negative class**, and another name for the class that we denote with **one is the positive class**. So zero we denote as the benign tumor, and one, positive class we denote a malignant tumor. The assignment of the two classes, spam not spam and so on.

$y \in \{0, 1\}$

- 0: "Negative Class" (e.g., benign tumor)
- 1: "Positive Class" (e.g., malignant tumor)

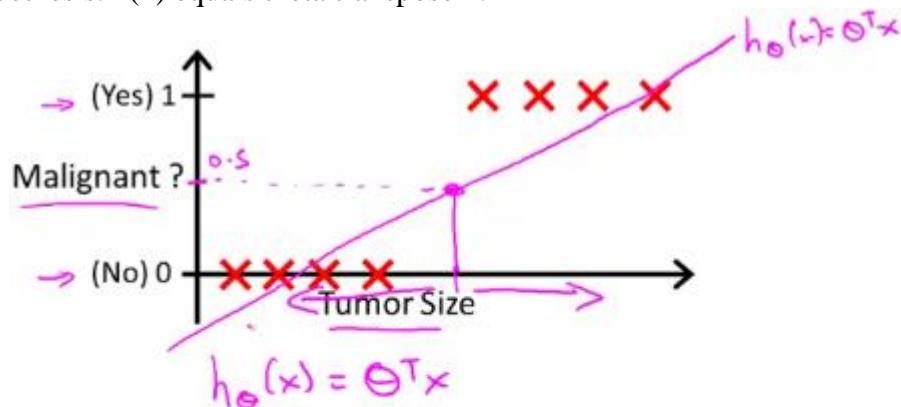
The assignment of the two classes to positive and negative to zero and one is somewhat arbitrary and it doesn't really matter but often there is this intuition that a **negative class is conveying the absence of something** like the absence of a malignant tumor. Whereas one the **positive class is conveying the presence of something that we may be looking for**, but the definition of which is negative and which is positive is somewhat arbitrary and it doesn't matter that much.

For now we're going to start with classification problems with just two classes zero and one. Later one we'll talk about multi class problems as well where therefore  $y$  may take on four values **zero, one, two, and three**. This is called a **multiclass classification** problem. But for the next few videos, let's start with the two class or the **binary classification** problem and we'll worry about the multiclass setting later.

$$\rightarrow y \in \{0, 1, 2, 3\}$$

So how do we develop a classification algorithm?

Here's an example of a training set for a classification task for classifying a tumor as malignant or benign. And notice that malignancy takes on only two values, zero or no, one or yes. So one thing we could do given this training set is to apply the algorithm that we already know, linear regression to this data set and just try to fit the straight line to the data. So if you take this training set and fit a straight line to it, maybe you get a hypothesis that looks like that, right. So that's my hypothesis.  $h(x)$  equals  $\theta^T x$ .



→ Threshold classifier output  $h_\theta(x)$  at 0.5:

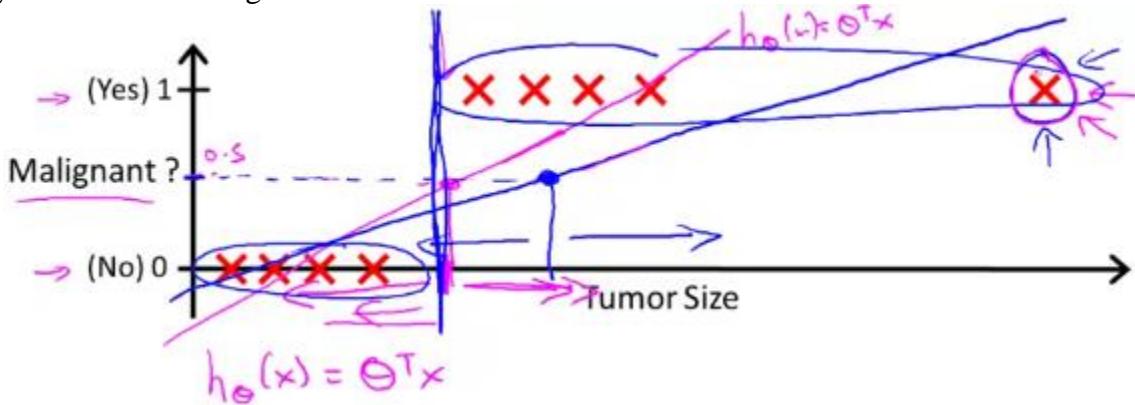
→ If  $h_\theta(x) \geq 0.5$ , predict "y = 1"

If  $h_\theta(x) < 0.5$ , predict "y = 0"

If you want to make predictions, one thing you could try doing is then threshold, the classifier outputs at 0.5 that is at a vertical axis value 0.5 and if the hypothesis outputs a value that is

greater than equal to 0.5 you can take  $y = 1$ . If it's less than 0.5 you can take  $y=0$ . Let's see what happens if we do that. So 0.5 and so that's where the threshold is and that's using linear regression this way. **Everything to the right of this point we will end up predicting as the positive cross.** Because the output values is greater than 0.5 on the vertical axis and **everything to the left of that point we will end up predicting as a negative value.** In this particular example, it looks like linear regression is actually doing something reasonable even though this is a classification task we're interested in. But now let's try changing the problem a bit.

Let me extend out the horizontal access a little bit and let's say we got one more training example way out there on the right.



→ Threshold classifier output  $h_\theta(x)$  at 0.5:

→ If  $h_\theta(x) \geq 0.5$ , predict "y = 1"

If  $h_\theta(x) < 0.5$ , predict "y = 0"

Notice that that additional training example, this one out here, it doesn't actually change anything, right. Looking at the training set it's pretty clear what a good hypothesis is. Is that well everything to the right of somewhere around here, to the right of this we should predict this positive. Everything to the left we should probably predict as negative because from this training set, it looks like all the tumors larger than a certain value around here are malignant, and all the tumors smaller than that are not malignant, at least for this training set.

But once we've added that extra example over here, if you now run linear regression, you instead get a straight line fit to the data. That might maybe look like this. And if you know threshold hypothesis at 0.5, you end up with a threshold that's around here, so that everything to the right of this point you predict as positive and everything to the left of that point you predict as negative.

And this seems a pretty bad thing for linear regression to have done, right, because you know these are our positive examples, these are our negative examples. It's pretty clear we really should be separating the two somewhere around there, but somehow by adding one example way out here to the right, this example really isn't giving us any new information. I mean, there should be no surprise to the learning algorithm that the example way out here turns out to be

malignant. But somehow having that example out there caused linear regression to change its straight-line fit to the data from this magenta line out here to this blue line over here, and caused it to give us a worse hypothesis. So, applying linear regression to a classification problem often isn't a great idea. In the first example, before I added this extra training example, previously linear regression was just getting lucky and it got us a hypothesis that worked well for that particular example, but usually applying linear regression to a data set, you might get lucky but often it isn't a good idea. So I wouldn't use linear regression for classification problems.

Here's one other funny thing about what would happen if we were to use linear regression for a classification problem.

Classification:  $y = 0 \text{ or } 1$

$$h_{\theta}(x) \text{ can be } > 1 \text{ or } < 0$$

Logistic Regression:  $0 \leq h_{\theta}(x) \leq 1$

( Classification )

For classification we know that  $y$  is either zero or one. But if you are using linear regression where the hypothesis can output values that are much larger than one or less than zero even if all of your training examples have labels  $y$  equals zero or one. And it seems kind of strange that even though we know that the labels should be zero, one it seems kind of strange if the algorithm can output values much larger than one or much smaller than zero.

So what we'll do in the next few videos is develop an algorithm called **logistic regression**, which has the property that the output, the predictions of logistic regression are always between zero and one, and doesn't become bigger than one or become less than zero. And by the way, logistic regression is, and we will use it as a **classification algorithm**, is some, maybe sometimes confusing that the term regression appears in this name even though logistic regression is actually a classification algorithm. But that's just a name it was given for historical reasons. So don't be confused by that logistic regression is actually a classification algorithm that we apply to settings where the label  $y$  is discrete value, when it's either zero or one.

So hopefully you now know why, if you have a classification problem, using linear regression isn't a good idea. In the next video, we'll start working out the details of the logistic regression algorithm.

## Hypothesis Representation

Let's start talking about logistic regression. In this video, I'd like to show you the **hypothesis representation**. That is, **what is the function we're going to use to represent our hypothesis** when we have a classification problem?

Earlier, we said that we would like our classifier to output values that are between 0 and 1. So we'd like to come up with a hypothesis that satisfies this property, that is, predictions are maybe between 0 and 1.

### Logistic Regression Model

$$\text{Want } \underline{0 \leq h_{\theta}(x) \leq 1}$$

When we were using linear regression, this was the form of a hypothesis, where  $h(x)$  is theta transpose  $x$ .

$$h_{\theta}(x) = \theta^T x$$

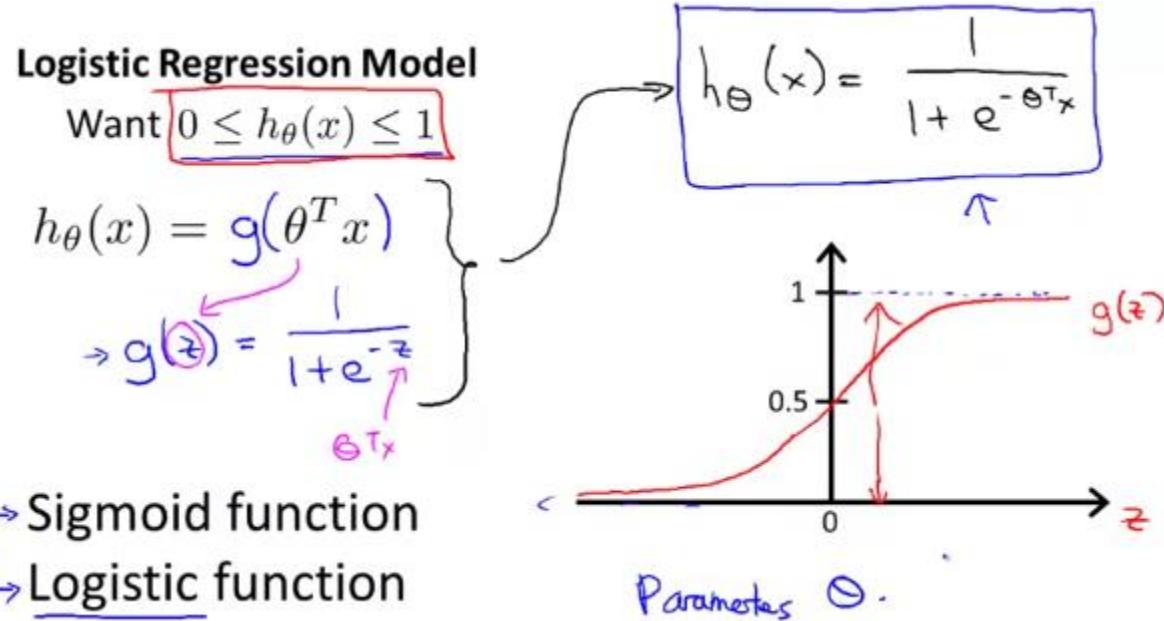
For logistic regression, I'm going to modify this a little bit and make the hypothesis  $g$  of theta transpose  $x$ .

$$h_{\theta}(x) = g(\theta^T x)$$

Where I'm going to define the function  $g$  as follows.  **$g(z)$** ,  $z$  is a real number, is equal to one over one plus  $e$  to the negative  $z$ .

$$g(z) = \frac{1}{1+e^{-z}}$$

This is called the **sigmoid function**, or the **logistic function**, and the term logistic function, that's what gives rise to the name **logistic regression**. And by the way, the terms sigmoid function and logistic function are basically synonyms and mean the same thing. So the two terms are basically interchangeable, and either term can be used to refer to this function  $g$ . And if we take these two equations and put them together, then here's just an alternative way of writing out the form of my hypothesis.



I'm saying that **h(x)** is **1 over 1 plus e to the negative theta transpose x**. And all I've done is I've taken this variable z, z here is a real number, and plugged in theta transpose x. So I end up with theta transpose x in place of z there.

Lastly, let me show you what the sigmoid function looks like. We're gonna plot it on this figure here. The sigmoid function,  $g(z)$ , also called the logistic function, it looks like this. It starts off near 0 and then it rises until it crosses 0.5 at the origin, and then it flattens out again like so.

So **that's what the sigmoid function looks like**. And you notice that the sigmoid function, while it asymptotes at one and asymptotes at zero, as a z, the horizontal axis is z. As z goes to minus infinity,  $g(z)$  approaches zero. And as  $g(z)$  approaches infinity,  $g(z)$  approaches one. And so because  $g(z)$  output values are between zero and one, we also have that  $h(x)$  must be between zero and one.

Finally, given this hypothesis representation, **what we need to do, as before, is fit the parameters theta to our data**. So given a training set **we need to pick a value for the parameters theta and this hypothesis will then let us make predictions**.

We'll talk about a learning algorithm later for fitting the parameters theta, but **first let's talk a bit about the interpretation of this model**. Here's how I'm going to interpret the output of my hypothesis,  $h(x)$ . When my hypothesis outputs some number, I am going to **treat that number as the estimated probability that y is equal to one** on a new input, example x.

## Interpretation of Hypothesis Output

$$h_{\theta}(x)$$

$h_{\theta}(x)$  = estimated probability that  $y = 1$  on input  $x$

Example: If  $\underline{x} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1 \\ \text{tumorSize} \end{bmatrix}$

$$\underline{h_{\theta}(x) = 0.7} \quad \underline{y=1}$$

Tell patient that 70% chance of tumor being malignant

$$\underline{h_{\theta}(x) = P(y=1|x; \theta)}$$

$$\underline{y = 0 \text{ or } 1}$$

"probability that  $y = 1$ , given  $x$ , parameterized by  $\theta$ "

$$\rightarrow P(y=0|\underline{x}; \theta) + P(y=1|\underline{x}; \theta) = 1$$

$$\rightarrow P(\underline{y=0|x; \theta}) = 1 - P(y=1|x; \theta)$$

Here's what I mean, here's an example. Let's say we're using the tumor classification example, so we may have a **feature vector  $x$** , which is this,  **$x$  zero equals one as always**. And then one feature is the size of the tumor.

Suppose I have a patient come in and they have some tumor size and **I feed their feature vector  $x$  into my hypothesis**. And suppose **my hypothesis outputs the number 0.7**. I'm going to interpret my hypothesis as follows. I'm gonna say that this hypothesis is telling me that **for a patient with features  $x$ , the probability that  $y$  equals 1 is 0.7**. In other words, I'm going to tell my patient that the tumor, sadly, has a 70 percent chance, or a 0.7 chance of being malignant. To write this out slightly more formally, or to write this out in math, I'm going to interpret my hypothesis output as  **$P(y=1 | x; \theta)$** . So for those of you that are familiar with probability, this equation may make sense. If you're a little less familiar with probability, then here's how I read this expression. This is the "probability that  $y$  is equal to one given  $x$ , given that my patient has features  $x$ , so given my patient has a particular tumor size represented by my features  $x$ . And this probability is parameterized by theta". So I'm basically going to **count on my hypothesis to give me estimates of the probability that  $y$  is equal to 1**. Now, since this is a classification task, we know that  $y$  must be either 0 or 1, right? Those are the only two values that  $y$  could possibly take on, either in the training set or for new patients that may walk into my office, or into the doctor's office in the future. So given  $h(x)$ , we can therefore compute the probability that  $y = 0$  as well.

Concretely, because  $y$  must be either 0 or 1 we know that the probability of  $y = 0$  plus the probability of  $y = 1$  must add up to 1. This first equation looks a little bit more complicated. It's basically saying that probability of  $y=0$  for a particular patient with features  $x$ , and given our parameters theta. Plus the probability of  $y=1$  for that same patient with features  $x$  and given theta parameters theta must add up to one. If this equation looks a little bit complicated, feel free to mentally imagine it without that  $x$  and theta. And this is just saying that the probability of  $y$

equals zero plus the probability of  $y$  equals one, must be equal to one. And we know this to be true because  $y$  has to be either zero or one, and so the chance of  $y$  equals zero, plus the chance that  $y$  is one, those two must add up to one. And so if you just take this term and move it to the right hand side, then you end up with this equation that says probability that  $y$  equals zero is 1 minus probability of  $y$  equals 1, and thus if our hypothesis  $h(x)$  gives us that term you can therefore quite simply compute the probability or compute the estimated probability that  $y$  is equal to 0 as well.

So, **you now know what the hypothesis representation is for logistic regression** and we're seeing what the **mathematical formula is, defining the hypothesis for logistic regression**.

In the next video, I'd like to try to give you **better intuition about what the hypothesis function looks like**. And I wanna tell you about something called the **decision boundary**. And we'll look at some visualizations together to try to get a better sense of what this hypothesis function of logistic regression really looks like.

## Decision Boundary

In the last video, we talked about the hypothesis representation for **logistic regression**.

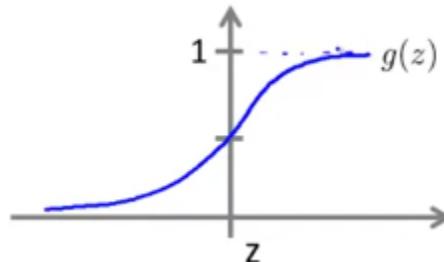
What I'd like to do now is tell you about something called the **decision boundary**, and this will give us **a better sense of what the logistic regressions hypothesis function is computing**.

To recap, this is what we wrote out last time, where we said that the hypothesis is represented as  **$h(x)$  equals  $g(\theta^T x)$** , where  **$g$  is this function called the sigmoid function**, which looks like this. It slowly increases from zero to one, asymptoting at one.

### Logistic regression

$$\rightarrow h_{\theta}(x) = g(\theta^T x)$$

$$\rightarrow g(z) = \frac{1}{1+e^{-z}}$$



What I want to do now is **try to understand better when this hypothesis will make predictions that  $y$  is equal to 1** versus when it might make predictions that  $y$  is equal to 0. And understand better **what hypothesis function looks like particularly when we have more than one feature**.

Concretely, this hypothesis is outputting estimates of the probability that  $y$  is equal to one, given  $x$  and parameterized by theta. So if we wanted to predict is  $y$  equal to one or is  $y$  equal to zero, here's something we might do. Whenever the hypothesis outputs that the probability of  $y$  being one is greater than or equal to 0.5, so this means that if there is more likely to be  $y$  equals 1 than  $y$  equals 0, then let's predict  $y$  equals 1. And otherwise, if the probability, the estimated probability of  $y$  being over 1 is less than 0.5, then let's predict  $y$  equals 0.

## Logistic regression

$$\rightarrow h_{\theta}(x) = g(\theta^T x) = P(y=1|x; \theta)$$

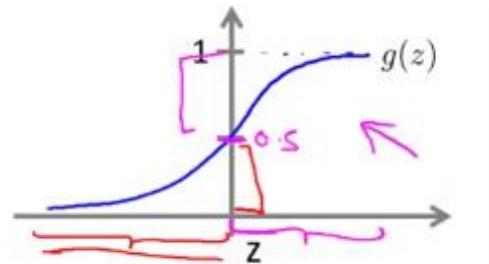
$$\rightarrow g(z) = \frac{1}{1+e^{-z}}$$

Suppose predict " $y = 1$ " if  $h_{\theta}(x) \geq 0.5$   
 $\theta^T x \geq 0$

predict " $y = 0$ " if  $h_{\theta}(x) < 0.5$

$$h_{\theta}(x) = g(\underline{\theta^T x})$$

$$\theta^T x < 0$$



$$g(z) \geq 0.5 \text{ when } z \geq 0$$

$$h_{\theta}(x) = g(\underline{\theta^T x}) \geq 0.5 \text{ whenever } \underline{\theta^T x} \geq 0$$

And I chose a greater than or equal to here and less than here if  $h$  of  $x$  is equal to 0.5 exactly, then you could predict positive or negative, but I probably created a loophole here, so we default maybe to predicting positive if  $h$  of  $x$  is 0.5, but that's a detail that really doesn't matter that much.

What I want to do is understand better **when is it exactly that  $h$  of  $x$  will be greater than or equal to 0.5, so that we'll end up predicting  $y$  is equal to 1**. If we look at this plot of the sigmoid function, we'll notice that the sigmoid function,  $g$  of  $z$  is greater than or equal to 0.5 **whenever  $z$  is greater than or equal to zero**. So in this half of the figure that  $g$  takes on values that are 0.5 and higher. This notch here, that's 0.5, and so when  $z$  is positive,  $g$  of  $z$ , the sigmoid function is greater than or equal to 0.5. Since the hypothesis for logistic regression is  $h$  of  $x$  equals  $g$  of  $\theta$  transpose  $x$ , this is therefore going to be greater than or equal to 0.5, **whenever theta transpose x is greater than or equal to 0**.

So what we're shown, right, because here **theta transpose x takes the role of  $z$** . So what we're shown is that a **hypothesis is gonna predict  $y$  equals 1 whenever theta transpose x is greater than or equal to 0**.

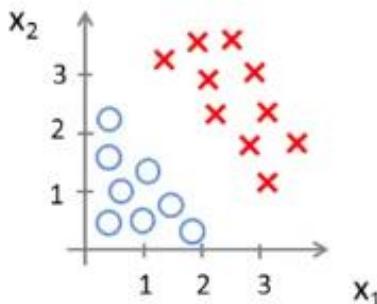
Let's now consider the other case of when a hypothesis will predict  $y$  is equal to 0. Well, by similar argument,  $h(x)$  is going to be less than 0.5 whenever  $g(z)$  is less than 0.5 because the range of values of  $z$  that cause  $g(z)$  to take on values less than 0.5, well, that's when  $z$  is negative. So **when  $g(z)$  is less than 0.5, a hypothesis will predict that  $y$  is equal to 0**. And by similar argument to what we had earlier,  $h(x)$  is equal to  $g$  of  $\theta$  transpose  $x$  and so we'll predict  $y$  equals 0 whenever this quantity  $\theta$  transpose  $x$  is less than 0.

To summarize what we just worked out, we saw that if we decide to predict whether  $y=1$  or  $y=0$  depending on whether the estimated probability is greater than or equal to 0.5, or whether less than 0.5, then that's the same as saying that when **we predict  $y=1$  whenever  $\theta$  transpose  $x$  is**

**greater than or equal to 0.** And we'll predict  $y$  is equal to 0 whenever theta transpose  $x$  is less than 0.

Let's use this to better understand how the hypothesis of logistic regression makes those predictions. Now, let's suppose we have a training set like that shown on the slide. And suppose a hypothesis is  $h(x)$  equals  $g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$ . We haven't talked yet about **how to fit the parameters of this model**. We'll talk about that in the next video. But **suppose** that via a procedure to be specified, we end up choosing the following values for the parameters.

### Decision Boundary



$$\Theta = \begin{bmatrix} -3 \\ 1 \\ 1 \end{bmatrix}$$

$$\rightarrow h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$$

$$-3 \quad // \quad 1 \quad 1$$

Let's say we choose  $\theta_0$  equals 3,  $\theta_1$  equals 1,  $\theta_2$  equals 1. So this means that my parameter vector is going to be  $\Theta = [-3, 1, 1]$ . So, when given this choice of my hypothesis parameters, **let's try to figure out where a hypothesis would end up predicting  $y$  equals one and where it would end up predicting  $y$  equals zero.**

Using the formulas that we were worked on in the previous slide, we know that  $y$  equals one is more likely, that is the probability that  $y$  equals one is greater than or equal to 0.5, whenever theta transpose  $x$  is greater than zero. And this formula that I just underlined,  $-3 + x_1 + x_2$ , is, of course, theta transpose  $x$  when theta is equal to this value of the parameters that we just chose.

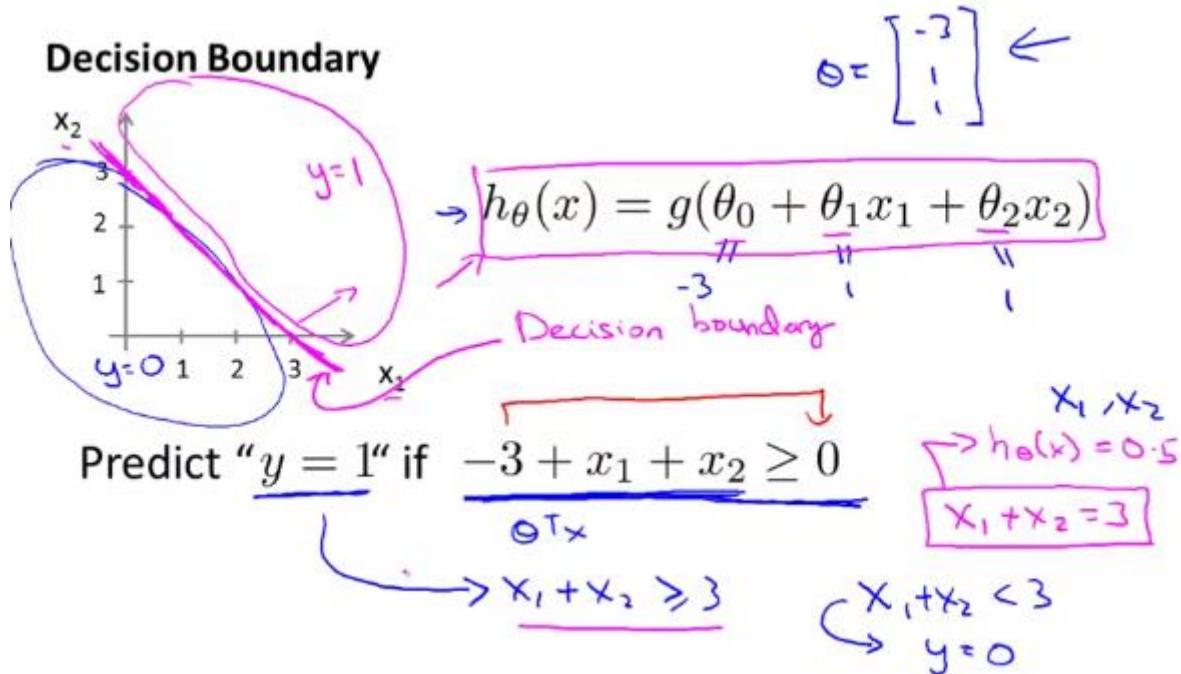
Predict " $y = 1$ " if  $\underline{-3 + x_1 + x_2} \geq 0$   
 $\Theta^T x$

So for any example, for any example which features  $x_1$  and  $x_2$  that satisfy this equation, that  $-3 + x_1 + x_2$  is greater than or equal to 0, our hypothesis will think that  $y$  equals 1 is more likely or predict that  $y$  is equal to 1.

We can also take -3 and bring this to the right and rewrite this as  $x_1 + x_2$  is greater than or equal to 3, so equivalently, we found that this hypothesis would predict  $y=1$  whenever  $x_1 + x_2$  is greater than or equal to 3.

Predict " $y = 1$ " if  $\underline{-3 + x_1 + x_2} \geq 0$   
 $\Theta^T x$   
 $\rightarrow x_1 + x_2 \geq 3$

Let's see what that means on the figure. If I write down the equation,  $x_1 + x_2 = 3$ , **this defines the equation of a straight line** and if I draw what that straight line looks like, it gives me the following line which passes through 3 and 3 on the  $x_1$  and the  $x_2$  axis.



So the part of the info space, the part of the  $x_1 x_2$  plane that corresponds to when  $x_1 + x_2$  is greater than or equal to 3, that's going to be this right half thing, that is everything to the up and everything to the upper right portion of this magenta line that I just drew. And so, the region where our hypothesis will predict  $y = 1$ , is this region, just really this huge region, this half space over to the upper right. And let me just write that down, I'm gonna call this the  $y = 1$  region.

And, in contrast, the region where  $x_1 + x_2$  is less than 3, that's when we will predict that  $y$  is equal to 0. And that corresponds to this region, that's really a half plane, but that region on the left is the region where our hypothesis will predict  $y = 0$ . I wanna give this line, this magenta line that I drew a name. **This line, there, is called the decision boundary.**

And concretely, this straight line,  $x_1 + x_2 = 3$ . That corresponds to the set of points, so that corresponds to the region where  $h(x)$  is equal to 0.5 exactly and the decision boundary that is this straight line, that's the line that separates the region where the hypothesis predicts  $y$  equals 1 from the region where the hypothesis predicts that  $y$  is equal to zero.

And just to be clear, the **decision boundary is a property of the hypothesis including the parameters  $\theta_0$ ,  $\theta_1$  and  $\theta_2$** . And in the figure I drew a training set, I drew a data set, in order to help the visualization. But even if we take away the data set this decision boundary and the region where we predict  $y = 1$  versus  $y = 0$ , that's a property of the hypothesis and of the parameters of the hypothesis and **not a property of the data set**.

Later on, of course, we'll talk about how to fit the parameters and there we'll end up using the training set, using our data to determine the value of the parameters. But once we have particular

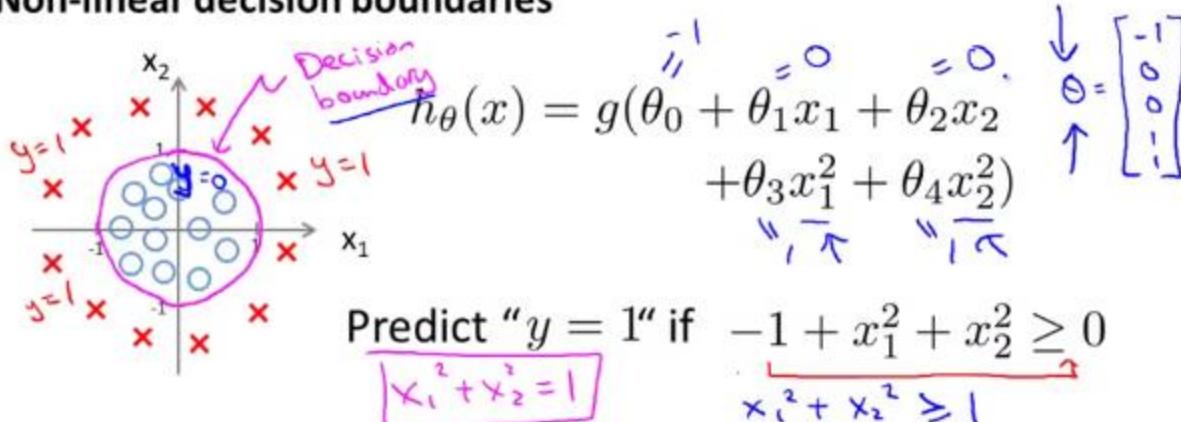
values for the parameters  $\theta_0$ ,  $\theta_1$  and  $\theta_2$  then that completely defines the decision boundary and we don't actually need to plot a training set in order to plot the decision boundary.

Let's now look at a more complex example where as usual, I have crosses to denote my positive examples and Os to denote my negative examples. Given a training set like this, how can I get logistic regression to fit the sort of data?

Earlier when we were talking about polynomial regression or when we're talking about linear regression, we talked about how we could add extra higher order polynomial terms to the features. And we can do the same for logistic regression.

Concretely, let's say my hypothesis looks like this where I've added two extra features,  $x_1$  squared and  $x_2$  squared, to my features. So that I now have five parameters,  $\theta_0$  through  $\theta_4$ . As before, we'll defer to the next video, our discussion on how to automatically choose values for the parameters  $\theta_0$  through  $\theta_4$ . But let's say that varied procedure to be specified, I end up choosing  $\theta_0$  equals minus one,  $\theta_1$  equals zero,  $\theta_2$  equals zero,  $\theta_3$  equals one and  $\theta_4$  equals one. What this means is that with this particular choice of parameters, my parameter vector theta looks like minus one, zero, zero, one, one.

### Non-linear decision boundaries



Following our earlier discussion, this means that my hypothesis will predict that  $y=1$  whenever  $-1 + x_1^2 + x_2^2$  is greater than or equal to 0. This is whenever theta transpose times my features is greater than or equal to zero. And if I take minus 1 and just bring this to the right, I'm saying that my hypothesis will predict that  $y$  is equal to 1 whenever  $x_1^2 + x_2^2$  is greater than or equal to 1.

So what does this decision boundary look like? Well, if you were to plot the curve for  $x_1$  squared plus  $x_2$  squared equals 1 Some of you will recognize that, that is the equation for circle of radius one, centered around the origin. So that is my decision boundary. And everything outside the circle, I'm going to predict as  $y=1$ . So out here is my  $y$  equals 1 region, we'll predict  $y$  equals 1 out here and inside the circle is where I'll predict  $y$  is equal to 0.

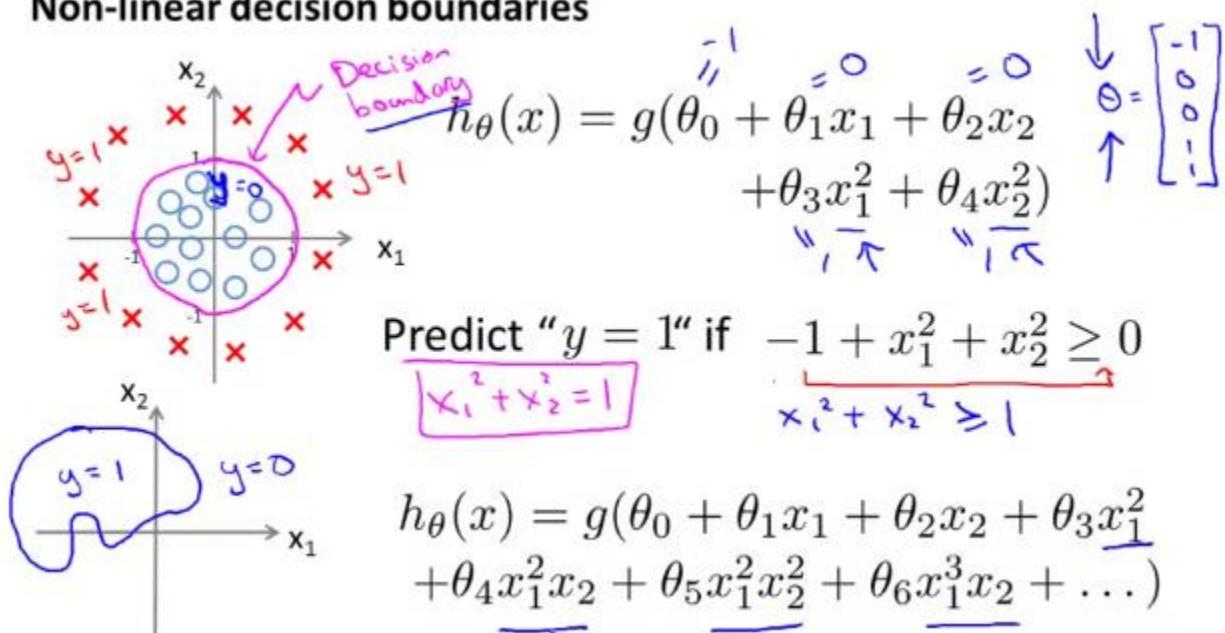
So by adding these more complex, or these polynomial terms to my features as well, I can get more complex decision boundaries that don't just try to separate the positive and negative examples in a straight line that I can get in this example, a decision boundary that's a circle.

Once again, the decision boundary is a property, not of the training set, but of the hypothesis under the parameters. So, so long as we're given my **parameter vector  $\theta$** , that **defines the decision boundary**, which is the circle.

But the training set is not what we use to define the decision boundary. The **training set may be used to fit the parameters  $\theta$** . We'll talk about how to do that later. But, once you have the parameters  $\theta$  that is what defines the decisions boundary. Let me put back the training set just for visualization.

And finally let's look at a more complex example.

### Non-linear decision boundaries



So can we come up with even more complex decision boundaries than this? **If I have even higher polynomial terms** so things like  $x_1^2$ ,  $x_1^2 x_2$ ,  $x_1^2 x_2^2$  and so on. And have much higher polynomials, then it's possible to show that you can get even more complex decision boundaries and logistic regression can be used to find decision boundaries that may, for example, be an ellipse like that or maybe a little bit different setting of the parameters maybe you can get instead a different decision boundary which may even look like some funny shape like that. Or for even more complete examples maybe you can also get this decision boundaries that could look like more complex shapes like that where everything in here you predict  $y = 1$  and everything outside you predict  $y = 0$ . So this higher order polynomial features you can a very complex decision boundaries.

So, with these visualizations, I hope that gives you a sense of what's the range of hypothesis functions we can represent using the representation that we have for logistic regression.

Now that we know what  $h(x)$  can represent, what I'd like to do next in the following video is talk about **how to automatically choose the parameters theta so that given a training set we can automatically fit the parameters to our data**.

## Lecture Note

### Decision Boundary

In order to get our discrete 0 or 1 classification, we can translate the output of the hypothesis function as follows:

$$h_{\theta}(x) \geq 0.5 \rightarrow y=1$$

$$h_{\theta}(x) < 0.5 \rightarrow y=0$$

The way our logistic function  $g$  behaves is that when its input is greater than or equal to zero, its output is greater than or equal to 0.5:

$$g(z) \geq 0.5 \text{ when } z \geq 0$$

Remember.

$$z=0, \quad e^0=1 \quad \Rightarrow g(z) = 1/2$$

$$z \rightarrow \infty, \quad e^{-\infty} \rightarrow 0 \quad \Rightarrow g(z) = 1$$

$$z \rightarrow -\infty, \quad e^{\infty} \rightarrow \infty \quad \Rightarrow g(z) = 0$$

So if our input to  $g$  is  $\theta^T x$  then that means:

$$h_{\theta}(x) = g(\theta^T x) \geq 0.5$$

$$\text{when } \theta^T x \geq 0$$

From these statements we can now say:

$$\theta^T x \geq 0 \Rightarrow y=1$$

$$\theta^T x < 0 \Rightarrow y=0$$

The **decision boundary** is the line that separates the area where  $y = 0$  and where  $y = 1$ . It is created by our hypothesis function.

**Example:**

$\theta = [5$  $-1$  $0]$  $y = 1 \text{ if } 5 + (-1)x_1 + 0x_2 \geq 0$  $5 - x_1 \geq 0$  $-x_1 \geq -5$  $x_1 \leq 5$ 

In this case, our decision boundary is a straight vertical line placed on the graph where  $x_1=5$ , and everything to the left of that denotes  $y = 1$ , while everything to the right denotes  $y = 0$ .

Again, the input to the sigmoid function  $g(z)$  (e.g.  $\theta^T$ ) doesn't need to be linear, and could be a function that describes a circle (e.g.  $z = \theta_0 + \theta_1 x_1^2 + \theta_2 x_2^2$ ) or any shape to fit our data.

## Logistic Regression Model

### Cost Function

In this video, we'll talk about how to fit the parameters of theta for logistic regression. In particular, I'd like to define the optimization objective, or the cost function that we'll use to fit the parameters.

Here's the supervised learning problem of fitting a logistic regression model.

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

$$\underbrace{m \text{ examples}}_{\left[ h_{\theta}(x) = \frac{1}{1 + e^{-\theta^T x}} \right]} \quad x \in \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n+1} \quad \underbrace{x_0 = 1, y \in \{0, 1\}}$$

### How to choose parameters $\theta$ ?

We have a training set of  $m$  training examples and as usual, each of our examples is represented by a feature vector that's  $n$  plus one dimensional, and as usual we have  $x_0$  equals one. First feature or a zero feature is always equal to one. And because this is a classification problem, our training set has a property that every label  $y$  is either 0 or 1. This is a hypothesis, and the

parameters of this hypothesis is this  $\theta$  over here. **And the question that I want to talk about is given this training set, how do we choose, or how do we fit the parameter's theta?**

Back **when we were developing the linear regression model**, we used the following cost function. I've written this slightly differently where instead of 1 over 2m, I've taken a one-half and put it inside the summation instead.

### Cost function

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

Now I want to use an alternative way of writing out this cost function. Which is that instead of writing out this square of return here, let's write in here **cost( $\mathbf{h}(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ )** and I'm going to define that term **cost( $\mathbf{h}(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ )** to be equal to this, just **equal to this one-half of the squared error**. So now we can see more clearly that the cost function is a sum over my training set, which is 1 over n times the sum of my training set of this cost term here. And to simplify this equation a little bit more, **it's going to be convenient to get rid of those superscripts**. So just define cost of  $\mathbf{h}$  of  $\mathbf{x}$  comma  $\mathbf{y}$  to be equal to one half of this squared error.

### Cost function

$$\rightarrow \text{Linear regression: } J(\theta) = \boxed{\frac{1}{m} \sum_{i=1}^m \frac{1}{2} (h_\theta(x^{(i)}) - y^{(i)})^2}$$

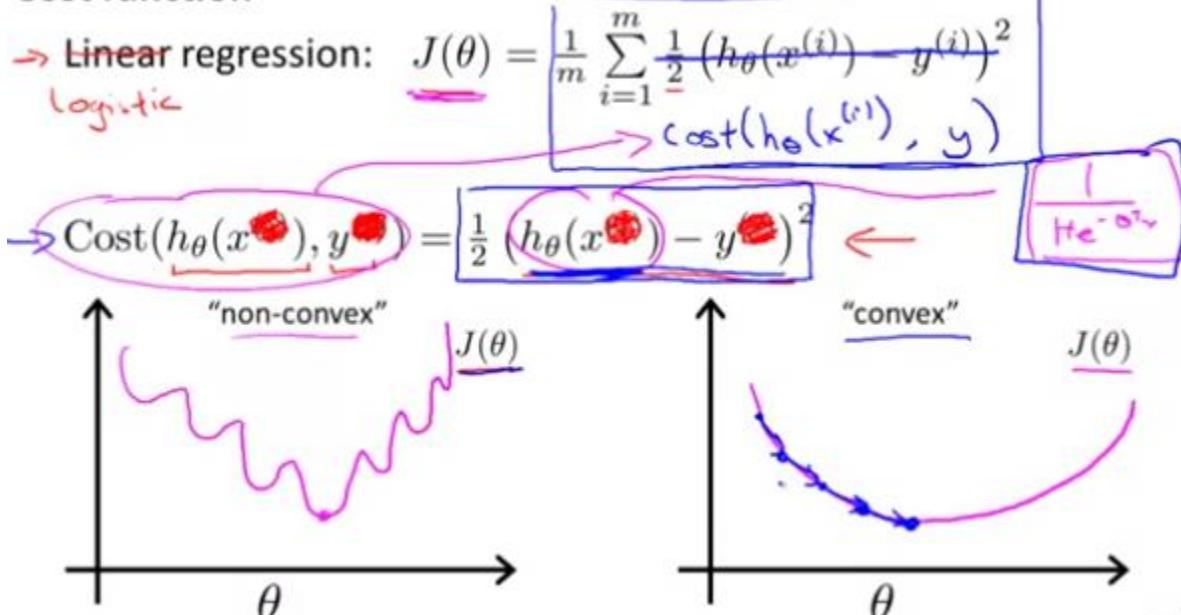
**cost( $h_\theta(\mathbf{x}^{(i)}, \mathbf{y})$ )**

$$\rightarrow \text{Cost}(h_\theta(x^{(i)}), y^{(i)}) = \frac{1}{2} (h_\theta(x^{(i)}) - y^{(i)})^2$$

And interpretation of this cost function is that, **this is the cost I want my learning algorithm to have to pay if it outputs that value**, if its **prediction is  $\mathbf{h}(\mathbf{x})$ , and the actual label was  $\mathbf{y}$** . So just cross off the superscripts, right, and no surprise **for linear regression** the cost we've defined is that or the **cost of this is** that is **one-half times the square difference between what I predicted and the actual value that we observed for  $\mathbf{y}$** .

Now this cost function worked fine for linear regression. But here, we're interested in logistic regression. If we could minimize this cost function that is plugged into  $J$  here, that will work okay. But it turns out that if we use this particular cost function, **this would be a non-convex function of the parameter's theta**. Here's what I mean by non-convex. Have some cross function  $j$  of  $\theta$  and for logistic regression, this function  $h$  here has a nonlinearity, this is one over one plus  $e$  to the negative  $\theta$  transpose  $x$ . So **this is a pretty complicated nonlinear function**.

## Cost function



And if you take the **sigmoid function, plug it in here**, and then take this cost function and plug it in there and then plot what  $j$  of  $\theta$  looks like, you find that  $j$  of  $\theta$  can look like a function that's like this, **with many local optima**. And the formal term for this is that this is a **non-convex function**. And you can kind of tell, if you were to run gradient descent on this sort of function it is not guaranteed to converge to the global minimum.

Whereas in contrast what we would like is to have a cost function  $j$  of  $\theta$  that is convex, that is a single bow-shaped function that looks like this so that if you run gradient descent we would be guaranteed that gradient descent would converge to the global minimum. And the problem with using this great cost function is that because of this very nonlinear sigmoid function that appears in the middle here, **J of theta ends up being a non-convex function if you were to define it as a square cost function.**

So what we'd like to do is to, instead of **come up with a different cost function** that is convex, and **so that we can apply a great algorithm, like gradient descent and be guaranteed to find the global minimum.**

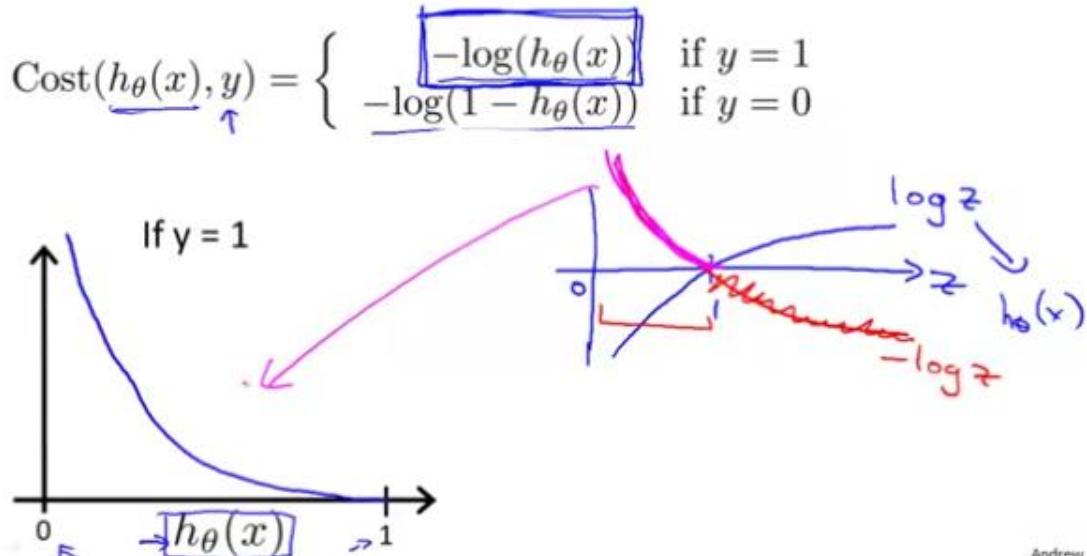
Here's the cost function that we're going to use for logistic regression. We're going to say that **the cost, or the penalty that the algorithm pays**, if it upwards the value of  $h(x)$ , so if this is some number like 0.7, it predicts the value  $h(x)$ , and the actual cost label turns out to be  $y$ , the cost is going to be **-log( $h(x)$ ) if  $y = 1$**  and **-log(1 -  $h(x)$ ) if  $y = 0$** .

## Logistic regression cost function

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

This looks like a pretty complicated function, but let's plot this function to gain some intuition about what it's doing.

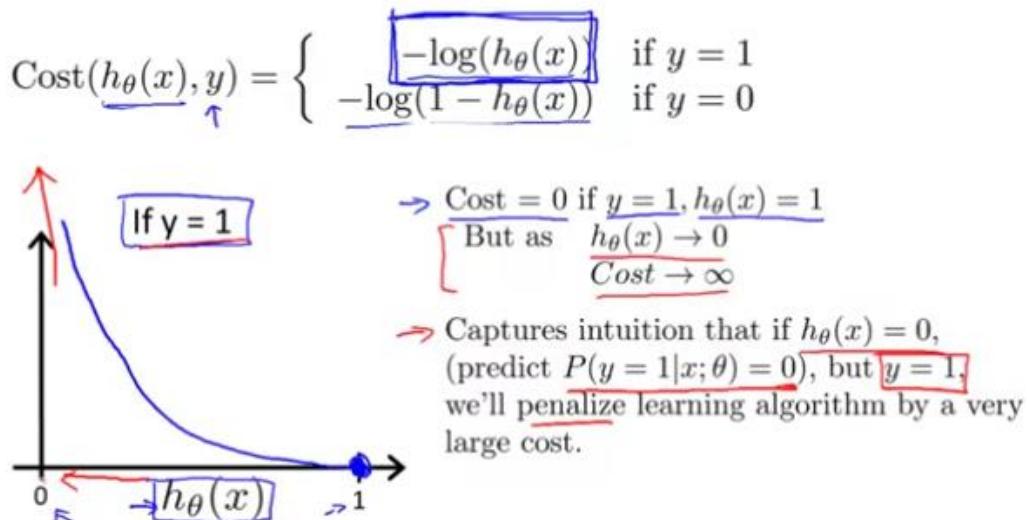
### Logistic regression cost function



Let's start off with the case of  $y = 1$ . If  $y = 1$ , then the cost function is  $-\log(h(x))$ . And if we plot that, so let's say that the horizontal axis is  $h(x)$ , so we know that a hypothesis is going to output a value between 0 and 1. Right, so  $h(x)$ , that varies between 0 and 1. If you plot what this cost function looks like, you find that it looks like this. One way to see why the plot looks like this is because if you were to plot  $\log z$  with  $z$  on the horizontal axis, then that looks like that. And it approaches minus infinity, right? So this is what the log function looks like. And this is 0, this is 1. Here,  $z$  is of course playing the role of  $h(x)$ . And so  $-\log z$  will look like this. Just flipping the sign, minus  $\log z$ , and we're interested only in the range of when this function goes between zero and one, so get rid of that. And so we're just left with, you know, this part of the curve, and that's what this curve on the left looks like.

Now, this cost function has a few interesting and desirable properties.

### Logistic regression cost function

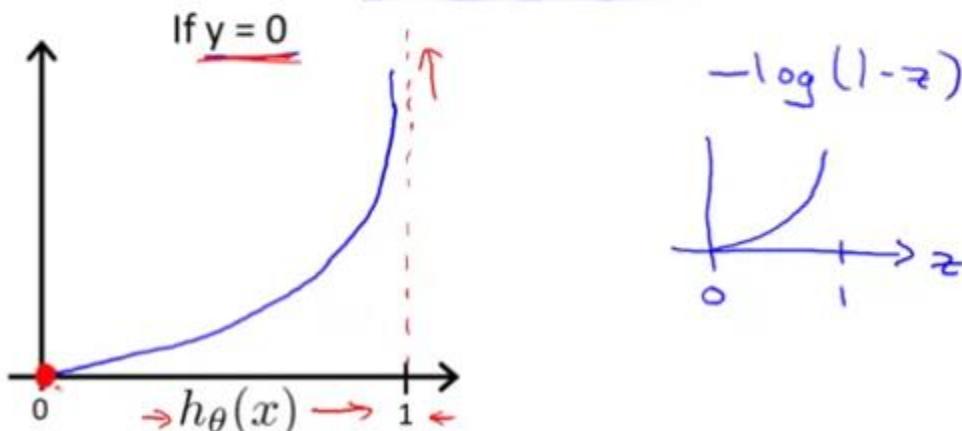


- First, you notice that if  $y$  is equal to 1 and  $h(x)$  is equal to 1, in other words, if the hypothesis exactly predicts  $h$  equals 1 and  $y$  is exactly equal to what it predicted, then the cost = 0 right? That corresponds to the curve doesn't actually flatten out. The curve is still going.
- First, notice that if  $h(x) = 1$ , if that hypothesis predicts that  $y = 1$  and if indeed  $y = 1$  then the cost = 0 that corresponds to this point down here, right? If  $h(x) = 1$  and we're only considering the case of  $y = 1$  here. But if  $h(x) = 1$  then the cost is down here, is equal to 0. And that's where we'd like it to be because if we correctly predict the output  $y$ , then the cost is 0.
- But now notice also that as  $h(x)$  approaches 0, so as  $h(x)$  the output of a hypothesis approaches 0, the cost blows up and it goes to infinity. And what this does is this captures the intuition that if a hypothesis you know outputs 0, that's like saying a hypothesis saying the chance of  $y$  equals 1 is equal to 0. It's kinda like our going to our medical patients and saying the probability that you have a malignant tumor, the probability that  $y=1$ , is zero. So, it's like absolutely impossible that your tumor is malignant.
- But if it turns out that the tumor, the patient's tumor, actually is malignant, so if  $y$  is equal to one, even after we told them, that the probability of it happening is zero. So it's absolutely impossible for it to be malignant. **But if we told them this with that level of certainty and we turn out to be wrong, then we penalize the learning algorithm by a very, very large cost.** And that's captured by having this **cost go to infinity if  $y$  equals 1 and  $h(x)$  approaches 0**.

This slide considered the case of  $y$  equals 1. **Let's look at what the cost function looks like for  $y$  equals 0.**

### Logistic regression cost function

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



If  $y$  is equal to 0, then the cost looks like this, it looks like this expression over here, and if you plot the function,  **$-\log(1-z)$** , what you get is the cost function actually looks like this. So it goes from 0 to 1, something like that, and so if you plot the cost function for the case of  $y$  equals 0, you find that it looks like this. And what this curve does is it now goes up and it goes to

plus infinity as  $h$  of  $x$  goes to 1 because as I was saying, that if  $y$  turns out to be equal to 0, but we predicted that  $y$  is equal to 1 with almost certainly, probability 1, then we end up paying a very large cost. And conversely, if  $h$  of  $x$  is equal to 0 and  $y$  equals 0, then the hypothesis predicted  $y$  of  $x$  is equal to 0, and it turns out  $y$  is equal to 0, so at this point, the cost function is going to be 0.

In this video, we defined the **cost function for a single training example**. The topic of convexity analysis is beyond the scope of this course, but it is possible to show that with a particular choice of cost function, this will give us a convex optimization problem. Overall cost function  $J$  of  $\theta$  will be convex and local optima free.

In the next video we're gonna **take these ideas of the cost function for a single training example and develop that further, and define the cost function for the entire training set**. And we'll also **figure out a simpler way to write it** than we have been using so far, and based on that **we'll work out gradient descent**, and **that will give us logistic regression algorithm**.

## Simplified Cost Function and Gradient Descent

In this video, we'll **figure out a slightly simpler way to write the cost function** than we have been using so far. And we'll **also figure out how to apply gradient descent to fit the parameters of logistic regression**. So, by the end of this, video you know how to implement a fully working version of logistic regression.

Here's our cost function for logistic regression.

### Logistic regression cost function

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

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

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

Our overall cost function is **1 over m times the sum over the training set of the cost of making different predictions on the different examples of labels  $y^{(i)}$** . And this is the cost of a single example that we worked out earlier. And just want to remind you that for classification problems in our training sets, and in fact even for examples, now that our training set  $y$  is always equal to 0 or 1, right? That's sort of part of the mathematical definition of  $y$ . Because  $y$  is either 0 or 1, we'll be able to come up with a simpler way to write this cost function. And in particular, rather than writing out this cost function on two separate lines with two separate cases, so  $y$  equals one and  $y$  equals zero, I'm going to show you a way to take these two lines and compress them into one equation. And this would make it more convenient to write out a cost function and derive gradient descent.

Concretely, we can write out the cost function as follows.

$$\text{Cost}(h_\theta(x), y) = -y \log(h_\theta(x)) - (1-y) \log(1-h_\theta(x)) \leftarrow$$

We say that **cost( $h(x)$ ,  $y$ )**, I'm gonna write this as **-y times log  $h(x)$  - (1-y) times log (1- $h(x)$ )**. And I'll show you in a second that this expression, now this equation, is an equivalent way, or more compact way, of writing out this definition of the cost function that we have up here. Let's see why that's the case.

### Logistic regression cost function

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

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

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

$$\rightarrow \text{Cost}(h_\theta(x), y) = -y \log(h_\theta(x)) - (1-y) \log(1 - h_\theta(x))$$

If  $y=1$ :  $\text{Cost}(h_\theta(x), y) = -\log h_\theta(x)$

If  $y=0$ :  $\text{Cost}(h_\theta(x), y) = -\log(1 - h_\theta(x))$

We know that there are only two possible cases.  $y$  must be 0 or 1.

- So let's suppose  $y$  equals one. If  $y$  is equal to 1, then this equation is saying that the cost is equal to, well if  $y$  is equal to 1, then this thing here is equal to 1. And 1 minus  $y$  is going to be equal to 0, right. So if  $y$  is equal to 1, then 1 minus  $y$  is 1 minus 1, which is therefore 0. So the second term gets multiplied by 0 and goes away. And we're left with only this first term, which is  $y$  times log, **-y times log ( $h(x)$ )**.  $y$  is 1 so that's equal to **-log  $h(x)$** . And this equation is exactly what we have up here for if  $y = 1$ .
- The other case is if  $y = 0$ . And if that's the case, then our writing of the cost function is saying that, well, if  $y$  is equal to 0, then this term here would be equal to zero. Whereas 1 minus  $y$ , if  $y$  is equal to zero would be equal to 1, because 1 minus  $y$  becomes 1 minus zero which is just equal to 1. And so the cost function simplifies to just this last term here, right? Because the first term over here gets multiplied by zero, and so it disappears, and so it's just left with this last term, which is **-log (1- $h(x)$ )**. And you can verify that this term here is just exactly what we had for when  $y$  is equal to 0.

So this shows that this definition for the cost is just a more compact way of taking both of these expressions, the cases  $y=1$  and  $y=0$ , and writing them in a more convenient form with just one line.

We can therefore write all our **cost functions** for **logistic regression** as follows.

### Logistic regression cost function

$$\begin{aligned} J(\theta) &= \frac{1}{m} \sum_{i=1}^m \text{Cost}(h_\theta(x^{(i)}), y^{(i)}) \\ &= -\frac{1}{m} \left[ \sum_{i=1}^m y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)})) \right] \end{aligned}$$

It is this 1 over m of the sum of these cost functions. And plugging in the definition for the cost that we worked out earlier, we end up with this. And we just put the minus sign outside. And why do we choose this particular function, while it looks like there could be other cost functions we could have chosen.

Although I won't have time to go into great detail of this in this course, this cost function can be derived from statistics using the principle of maximum likelihood estimation. Which is an idea in statistics for how to efficiently find parameters' data for different models. And it also has a nice property that it is convex. So this is the cost function that essentially everyone uses when fitting logistic regression models. If you don't understand the terms that I just said, if you don't know what the principle of maximum likelihood estimation is, don't worry about it. But it's just a deeper rationale and justification behind this particular cost function than I have time to go into in this class.

Given this cost function, in order to fit the parameters, what we're going to do then is try to find the parameters  $\theta$  that minimize  $J(\theta)$ . So if we try to minimize this, this would give us some set of parameters theta.

### Logistic regression cost function

$$\begin{aligned} J(\theta) &= \frac{1}{m} \sum_{i=1}^m \text{Cost}(h_\theta(x^{(i)}), y^{(i)}) \\ &= -\frac{1}{m} \left[ \sum_{i=1}^m y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)})) \right] \end{aligned}$$

To fit parameters  $\theta$ :

$$\min_{\theta} J(\theta) \quad \text{Get } \underline{\theta}$$

To make a prediction given new  $x$ :

$$\text{Output } \underline{h_\theta(x)} = \frac{1}{1+e^{-\theta^T x}} \quad \underline{\text{p}(y=1 | x; \theta)}$$

Finally,

- if we're given a new example with some set of features  $x$ ,
- we can then take the thetas that we fit to our training set and output our prediction as this.
- And just to remind you, the output of my hypothesis I'm going to interpret as the probability that  $y$  is equal to one, and given the input  $x$  and parameterized by theta. But just, you can think of this as just my hypothesis estimating the probability that  $y$  is equal to one.

So all that remains to be done is figure out how to actually minimize  $J(\theta)$  as a function of theta so that we can actually fit the parameters to our training set.

The way we're going to minimize the cost function is using gradient descent.

Here's our cost function and if we want to minimize it as a function of theta, here's our usual template for gradient descent where we repeatedly update each parameter by taking, updating it as itself minus learning rate alpha times this derivative term. If you know some calculus, feel free to take this term and try to compute the derivative yourself and see if you can simplify it to the same answer that I get. But even if you don't know calculus don't worry about it. If you actually compute this, what you get is this equation, and just write it out here. It's sum from i equals one through m of essentially the error times  $x^{(i)}$ . So if you take this partial derivative term and plug it back in here, we can then write out our gradient descent algorithm as follows.

### Gradient Descent

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

Want  $\min_{\theta} J(\theta)$ :

Repeat {

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

(simultaneously update all  $\theta_j$ )

$$\Theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} \quad \text{for } i=0 \dots n$$

$$h_\theta(x) = \theta^T x$$

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

Algorithm looks identical to linear regression!

And all I've done is I took the derivative term for the previous slide and plugged it in there. So if you have n features, you would have a parameter vector theta, which with parameters theta 0, theta 1, theta 2, down to theta n. And you will use this update to simultaneously update all of your values of theta. Now, if you take this update rule and compare it to what we were doing for linear regression. You might be surprised to realize that, well, this equation was exactly what we had for linear regression. In fact, if you look at the earlier videos, and look at the update rule, the Gradient Descent rule for linear regression, it looked exactly like what I drew here inside the blue box.

**So are linear regression and logistic regression different algorithms or not?** Well, this is resolved by observing that for logistic regression, what has changed is that the definition for this hypothesis has changed. So as whereas for linear regression, we had  $h(x)$  equals theta transpose X, now this definition of  $h(x)$  has changed. And is instead now one over one plus e to the negative transpose x. So even though the update rule looks cosmetically identical, because the definition of the hypothesis has changed, this is actually not the same thing as gradient descent for linear regression.

In an earlier video, when we were talking about gradient descent for linear regression, we had talked about how to monitor a gradient descent to make sure that it is converging. I usually apply that same method to logistic regression too to monitor a gradient descent, to make sure it's converging correctly. And hopefully, you can figure out how to apply that technique to logistic regression yourself. When implementing logistic regression with gradient descent, we have all of

these different parameter values, theta zero down to theta n that we need to update using this expression. And one thing we could do is have a for loop. So for i equals zero to n, or for i equals one to n plus one. So update each of these parameter values in turn. But of course rather than using a for loop, ideally we would also use a vectorized implementation.

So that a vectorized implementation can update all of these m plus one parameters all in one swoop. And to check your own understanding, you might see if you can figure out how to do the vectorized implementation with this algorithm as well.

So, now you know how to implement gradient descents for logistic regression. There was one last idea that we had talked about earlier for linear regression, which was feature scaling. We saw how feature scaling can help gradient descent converge faster for linear regression. The idea of feature scaling also applies to gradient descent for logistic regression. And yet we have features that are on very different scale, then applying feature scaling can also make grading descent run faster for logistic regression.

So that's it, you now know how to implement logistic regression and this is a very powerful, and probably the most widely used, classification algorithm in the world. And you now know how we get it to work for yourself.

## Advanced Optimization

In the last video, we talked about gradient descent for minimizing the cost function  $J(\theta)$  for logistic regression.

In this video, I'd like to tell you about some advanced optimization algorithms and some advanced optimization concepts. Using some of these ideas, we'll be able to get logistic regression to run much more quickly than it's possible with gradient descent. And this will also let the algorithms scale much better to very large machine learning problems, such as if we had a very large number of features.

Here's an alternative view of what gradient descent is doing. We have some cost function  $J$  and we want to minimize it. So what we need to is, we need to write code that can take as input the parameters  $\theta$  and can compute two things:

- i.  $J(\theta)$  and
- ii. these partial derivative terms for, you know, j equals 0, 1 up to n.

### Optimization algorithm

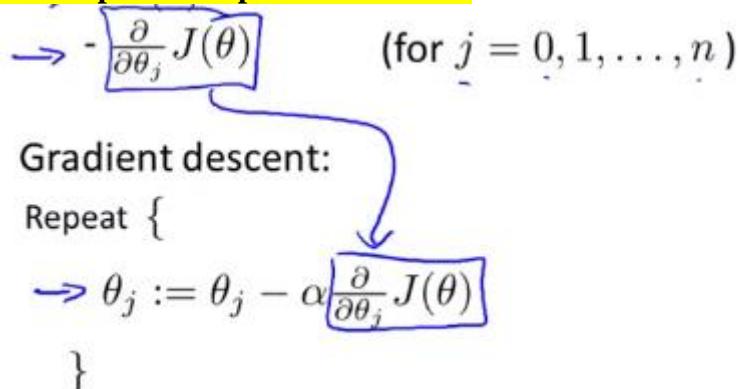
Cost function  $J(\theta)$ . Want  $\min_{\theta} J(\theta)$ .

Given  $\theta$ , we have code that can compute

$$\begin{aligned} \rightarrow & -J(\theta) \\ \rightarrow & -\frac{\partial}{\partial \theta_j} J(\theta) \quad (\text{for } j = 0, 1, \dots, n) \end{aligned}$$

Given code that can do these two things, what gradient descent does is it repeatedly performs the following update. Right?

So given the code that we wrote to compute these partial derivatives, **gradient descent plugs in here and uses that to update our parameters theta.**



So another way of thinking about gradient descent is that

- i. **we need to supply code to compute J of theta and these derivatives, and**
- ii. **then these get plugged into gradient descents, which can then try to minimize the function for us.**

For gradient descent, I guess technically you don't actually need code to compute the cost function J of theta. You only need code to compute the derivative terms. But if you think of your code as also monitoring convergence of some such, we'll just think of ourselves as providing code to compute both the cost function and the derivative terms.

So, having written code to compute these two things, one algorithm we can use is gradient descent. But **gradient descent isn't the only algorithm we can use.** And there are other algorithms, more advanced, more sophisticated ones, that, if we only provide them a way to compute these two things, then these are different approaches to optimize the cost function for us. So conjugate gradient **BFGS and L-BFGS are examples of more sophisticated optimization algorithms** that need a way to compute J of theta, and need a way to compute the derivatives, and can then use more sophisticated strategies than gradient descent to minimize the cost function. The details of exactly what these three algorithms are, is well beyond the scope of this course. And in fact you often end up spending, you know, many days, or a small number of weeks studying these algorithms. If you take a class and advance the numerical computing. But let me just tell you about some of their properties.

### Optimization algorithms:

- - Gradient descent
- └ - Conjugate gradient
- BFGS
- L-BFGS

### Advantages:

- No need to manually pick  $\alpha$
- Often faster than gradient descent.

### Disadvantages:

- More complex

These three algorithms have a number of **advantages.**

- i. One is that, with any of these algorithms you usually do not need to manually pick the learning rate alpha. So one way to think of these algorithms is that given is the way to compute the derivative and a cost function, you can think of these algorithms as having a clever inner-loop. And, in fact, they have a clever inner-loop called a line search algorithm that automatically tries out different values for the learning rate alpha and automatically picks a good learning rate alpha so that it can even pick a different learning rate for every iteration. And so then you don't need to choose it yourself.
- ii. These algorithms actually do more sophisticated things than just pick a good learning rate, and so they often end up converging much faster than gradient descent. These algorithms actually do more sophisticated things than just pick a good learning rate, and so they often end up converging much faster than gradient descent, but detailed discussion of exactly what they do is beyond the scope of this course.

In fact, I actually used to have used these algorithms for a long time, like maybe over a decade, quite frequently, and it was only, you know, a few years ago that I actually figured out for myself the details of what conjugate gradient, BFGS and L-BFGS do. So it is actually entirely possible to use these algorithms successfully and apply to lots of different learning problems without actually understanding the inner-loop of what these algorithms do.

If these algorithms have a **disadvantage**, I'd say that

- i. the main disadvantage is that they're quite a lot more complex than gradient descent.

And in particular, you probably should not implement these algorithms - conjugate gradient, L-BGFS, BFGS - yourself unless you're an expert in numerical computing. Instead, just as I wouldn't recommend that you write your own code to compute square roots of numbers or to compute inverses of matrices, for these algorithms also what I would recommend you do is just use a software library. So, you know, to take a square root what all of us do is use some function that someone else has written to compute the square roots of our numbers. And fortunately, Octave and the closely related language MATLAB - we'll be using that - Octave has a very good, has a pretty reasonable library implementing some of these advanced optimization algorithms. And so if you just use the built-in library, you know, you get pretty good results. I should say that there is a difference between good and bad implementations of these algorithms. And so, if you're using a different language for your machine learning application, if you're using C, C++, Java, and so on, you might want to try out a couple of different libraries to make sure that you find a good library for implementing these algorithms because there is a difference in performance between a good implementation of, you know, contour gradient or LPFGS versus less good implementation of contour gradient or LPFGS.

So now let's explain how to use these algorithms, I'm going to do so with an example.

**Example:**  $\min_{\theta} J(\theta)$

$$\rightarrow \theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} \quad \theta_1 = 5, \theta_2 = 5.$$

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

Let's say that **you have a problem with two parameters** equals  **$\theta_1$  and  $\theta_2$** . And let's say your cost function is  $J(\theta)$  equals theta one minus five squared, plus theta two minus five squared.

So with this cost function, you know, the value for  $\theta_1$  and  $\theta_2$ , if you want to minimize  $J(\theta)$  as a function of  $\theta$ . The value that minimizes it is going to be  $\theta_1 = 5$ , and  $\theta_2 = 5$ . Now, again, I know some of you know more calculus than others, but the derivatives of the cost function  $J$  turn out to be these two expressions. I've done the calculus.

$$\begin{aligned} \Rightarrow \frac{\partial}{\partial \theta_1} J(\theta) &= 2(\theta_1 - 5) \\ \Rightarrow \frac{\partial}{\partial \theta_2} J(\theta) &= 2(\theta_2 - 5) \end{aligned}$$

So if you want to apply one of the advanced optimization algorithms to minimize this cost function  $J$ . So, **if we didn't know the minimum was at 5, 5**, but if you want to have a cost function find the minimum numerically using something like gradient descent but preferably more advanced than gradient descent, what you would do is **implement an octave function** like this, so we implement a cost function, cost function theta function like that, and what this does is that it returns two arguments, the first  $J$ -val, is how we would compute the cost function  $J$ . And so this says  $J$ -val equals, you know, theta one minus five squared plus theta two minus five squared.

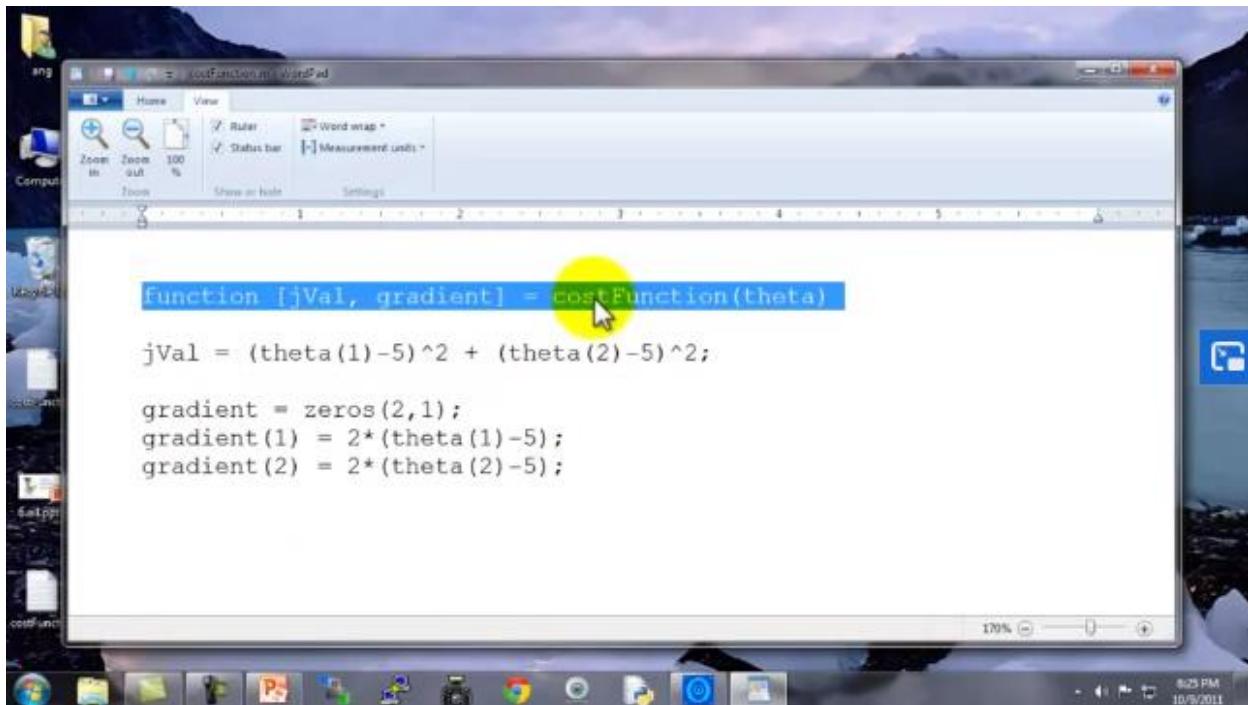
<p><b>Example:</b></p> $\rightarrow \theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} \quad \begin{matrix} \min_{\theta} J(\theta) \\ \theta_1 = 5, \theta_2 = 5 \end{matrix}$ $\rightarrow J(\theta) = (\theta_1 - 5)^2 + (\theta_2 - 5)^2$ $\rightarrow \frac{\partial}{\partial \theta_1} J(\theta) = 2(\theta_1 - 5)$ $\rightarrow \frac{\partial}{\partial \theta_2} J(\theta) = 2(\theta_2 - 5)$	<pre>function [jVal, gradient] = costFunction(theta)     jVal = (theta(1)-5)^2 + ... (theta(2)-5)^2;     gradient = zeros(2,1);     gradient(1) = 2*(theta(1)-5);     gradient(2) = 2*(theta(2)-5);  options = optimset('GradObj', 'on', 'MaxIter', '100'); initialTheta = zeros(2,1); [optTheta, functionVal, exitFlag] ... = fminunc(@costFunction, initialTheta, options);</pre>
--	---

So it's just computing this cost function over here. And the second argument that this function returns is gradient. **So gradient is going to be a two by one vector, and the two elements of the gradient vector correspond to the two partial derivative terms over here.** Having implemented this cost function, you would, you can then call the advanced optimization function called the **fminunc** - it stands for **function minimization unconstrained** in Octave -and the way you call this is as follows.

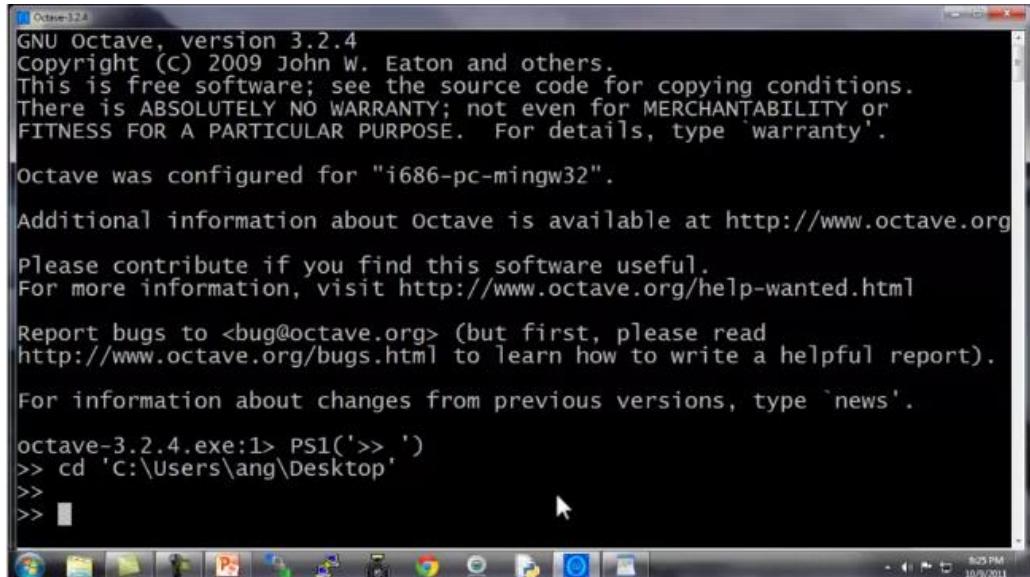
- You set a few **options**. This options is a data structure that stores the options you want. So grant up on, this sets the **gradient objective** parameter to on. It just means you are indeed going to provide a gradient to this algorithm.
- I'm going to set the **maximum number of iterations** to, let's say, one hundred.
- We're going give it an **initial guess for  $\theta$** . There's a **2 by 1 vector**.
- And then this command calls **fminunc**.
- This **@ symbol** presents a pointer to the cost function that we just defined up there. And if you call this, this will compute, you know, will use one of the more **advanced**

**optimization algorithms.** And if you want to think it as just like gradient descent. But automatically choosing the learning rate alpha so you don't have to do so yourself. But it will then attempt to use the sort of advanced optimization algorithms. Like gradient descent on steroids to try to find the optimal value of theta for you.

Let me actually show you what this looks like in Octave.



So I've written this cost function of theta, function exactly as we had it on the previous slide. It computes J-val which is the cost function. And it computes the gradient with the two elements being the partial derivatives of the cost function with respect to, you know, the two parameters, theta one and theta two. Now let's switch to my Octave window.

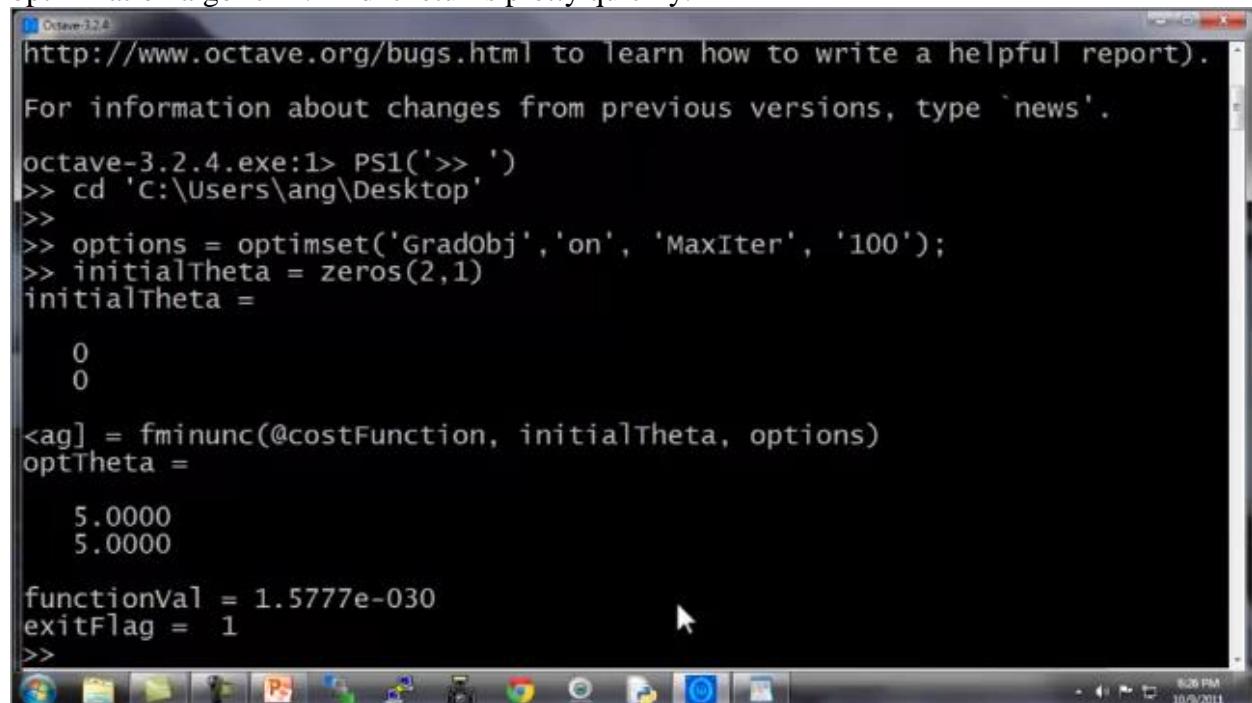


```

Octave-3.2.4
GNU Octave, version 3.2.4
Copyright (C) 2009 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. For details, type 'warranty'.
Octave was configured for "i686-pc-mingw32".
Additional information about octave is available at http://www.octave.org
Please contribute if you find this software useful.
For more information, visit http://www.octave.org/help-wanted.html
Report bugs to <bug@octave.org> (but first, please read
http://www.octave.org/bugs.html to learn how to write a helpful report).
For information about changes from previous versions, type 'news'.
octave-3.2.4.exe:1> PS1('>> ')
>> cd 'C:\Users\ang\Desktop'
>>
>> █

```

I'm gonna type in those commands I had just now. So, **options** equals **optimset**. This is the notation for setting my parameters on my options, for my optimization algorithm. Grant **option on, maxIter**, 100 so that says **100 iterations**, and I am going to provide the gradient to my algorithm. Let's say **initial theta** equals **zero's two by one**. So that's my initial guess for theta. And now I have optTheta, function val exit flag equals fminunc constraint. A pointer to the cost function and provide my initial guess. And the options like so. And if I hit enter this will run the optimization algorithm. And it returns pretty quickly.



```

http://www.octave.org/bugs.html to learn how to write a helpful report).

For information about changes from previous versions, type 'news'.

octave-3.2.4.exe:1> PS1('>> ')
>> cd 'C:\Users\ang\Desktop'
>>
>> options = optimset('GradObj','on', 'MaxIter', '100');
>> initialTheta = zeros(2,1)
initialTheta =
    0
    0

<ag> = fminunc(@costFunction, initialTheta, options)
optTheta =
    5.0000
    5.0000

functionVal = 1.5777e-030
exitFlag = 1
>> █

```

This funny formatting that's because my line, you know, my code wrapped around. So, this funny thing is just because my command line had wrapped around. But what this says is that numerically renders, you know, think of it as gradient descent on steroids, they found the optimal value of a theta is theta 1 equals 5, theta 2 equals 5, exactly as we're hoping for. The **function**

**value at the optimum is essentially 10 to the minus 30.** So that's essentially zero, which is also what we're hoping for. And the exit flag is 1, and this shows what the convergence status of this. And if you want you can do help fminunc to read the documentation for how to interpret the exit flag. But the exit flag let's you verify whether or not this algorithm thing has converged. **So that's how you run these algorithms in Octave.**

I should mention, by the way, that for the Octave implementation, this value of theta, your **parameter vector of theta**, must be in  $\mathbb{R}^d$  for  $d$  greater than or equal to 2. So if theta is just a real number. So, if it is not at least a two-dimensional vector or some higher than two-dimensional vector, this fminunc may not work, so and if in case you have a one-dimensional function that you use to optimize, you can look in the octave documentation for fminunc for additional details.

So, that's how we optimize our trial example of this simple quadratic cost function. How do we apply this to logistic regression? **In logistic regression we have a parameter vector theta**, and I'm going to use a mix of octave notation and sort of math notation. But I hope this explanation will be clear, but **our parameter vector theta comprises these parameters  $\theta_0$  through  $\theta_n$**  because octave indexes vectors using indexing from 1, you know,  $\theta_0$  is actually written  $\theta_1$  in octave,  $\theta_1$  is gonna be written  $\theta_2$  in octave, and that's going to be a written  $\theta_{n+1}$ , right? And that's because Octave indexes its vectors starting from index of 1 and so not the index of 0. So what we need to do then is **write a cost function** that captures the cost function **for logistic regression.**

theta = 
$$\begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$$
 theta(1) ←  
theta(2)  
theta(n+1)

```

function [jVal, gradient] = costFunction(theta)
    jVal = [code to compute J(θ)];
    gradient(1) = [code to compute  $\frac{\partial}{\partial \theta_0} J(\theta)$ ];
    gradient(2) = [code to compute  $\frac{\partial}{\partial \theta_1} J(\theta)$ ];
    :
    gradient(n+1) = [code to compute  $\frac{\partial}{\partial \theta_n} J(\theta)$ ];

```

Concretely, the cost function needs to return jval, which is you know jval, as you need some code to compute  $J(\theta)$  and we also need to give it the gradient. So, gradient 1 will be some code to compute the partial derivative with respect to  $\theta_0$ , the next partial derivative with respect to  $\theta_1$  and so on. Once again, this is gradient 1, gradient 2 and so on, rather than gradient 0, gradient 1 because octave indexes its vectors starting from one rather than from zero.

But **the main concept** I hope you take away from this slide is, that **what you need to do**, is **write a function that returns the cost function** and **returns the gradient**. And so in order to apply this to logistic regression or even to linear regression, if you want to use these optimization algorithms for linear regression, what you need to do is plug in the appropriate code to compute these things over here.

So, now you know how to use these advanced optimization algorithms. Because, using, because for these algorithms, you're using a sophisticated optimization library, it makes the just a little bit more opaque and so just maybe a little bit harder to debug. But because these algorithms often run much faster than gradient descent, often quite typically whenever I have a large machine learning problem, I will use these algorithms instead of using gradient descent. And with these ideas, hopefully, you'll be able to get logistic progression and also linear regression to work on much larger problems.

So, that's it for advanced optimization concepts. And **in the next and final video on Logistic Regression**, I want to tell you how to take the logistic regression algorithm that you already know about and make it work also **on multi-class classification** problems.

## Multiclass Classification: One-vs-all

In this video we'll talk about **how to get logistic regression to work for multiclass classification** problems. And in particular I want to tell you about **an algorithm called one-versus-all classification**.

What's a multiclass classification problem? Here are some examples.

### Multiclass classification

Email foldering/tagging: Work, Friends, Family, Hobby

$$\begin{array}{cccc} & \uparrow & \uparrow & \uparrow \\ y=1 & & y=2 & & y=3 & & y=4 \end{array}$$

Medical diagrams: Not ill, Cold, Flu

$$\begin{array}{ccc} y=1 & 2 & 3 \end{array}$$

Weather: Sunny, Cloudy, Rain, Snow

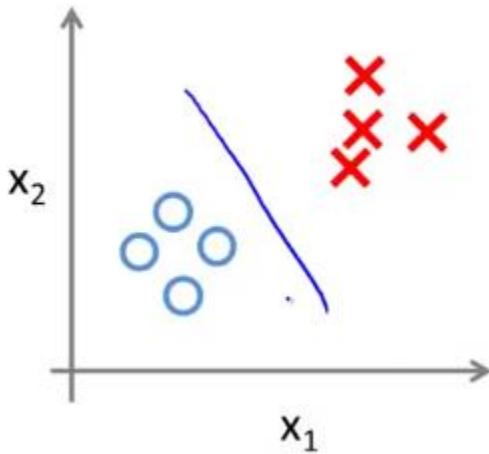
$$\begin{array}{cccc} y=1 & 2 & 3 & 4 \leftarrow \\ \cdot 0 & 1 & 2 & 3 \end{array}$$

Lets say you want a learning algorithm to automatically put your email into different folders or to automatically tag your emails so you might have different folders or different tags for work email, email from your friends, email from your family, and emails about your hobby. And so

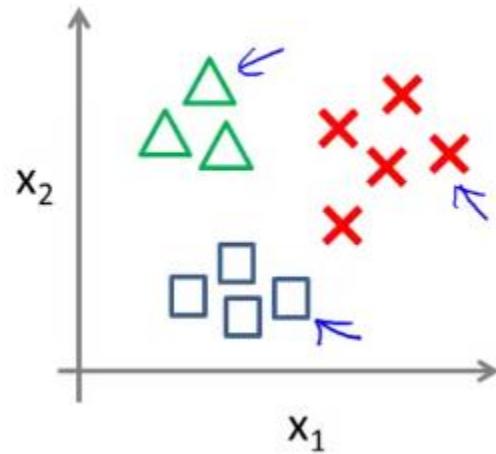
here we have a classification problem with **four classes** which we might assign to the classes  $y = 1$ ,  $y = 2$ ,  $y = 3$ , and  $y = 4$  too. And another example, for medical diagnosis, if a patient comes into your office with maybe a stuffy nose, the possible diagnosis could be that they're not ill. Maybe that's  $y = 1$ . Or they have a cold, 2. Or they have a flu. And a third and final example if you are using machine learning to classify the weather, you know maybe you want to decide that the weather is sunny, cloudy, rainy, or snow, or if it's gonna be snow, and so in all of these examples,  $y$  can take on a small number of discrete values, maybe one to three, one to four and so on, and these are multiclass classification problems. And by the way, it doesn't really matter whether we index is at 0, 1, 2, 3, or as 1, 2, 3, 4. I tend to index my classes starting from 1 rather than starting from 0, but either way we're off and it really doesn't matter.

Whereas previously for a binary classification problem, our data sets looked like this, for a multi-class classification problem our data sets may look like this where here I'm using three different symbols to represent our three classes.

**Binary classification:**

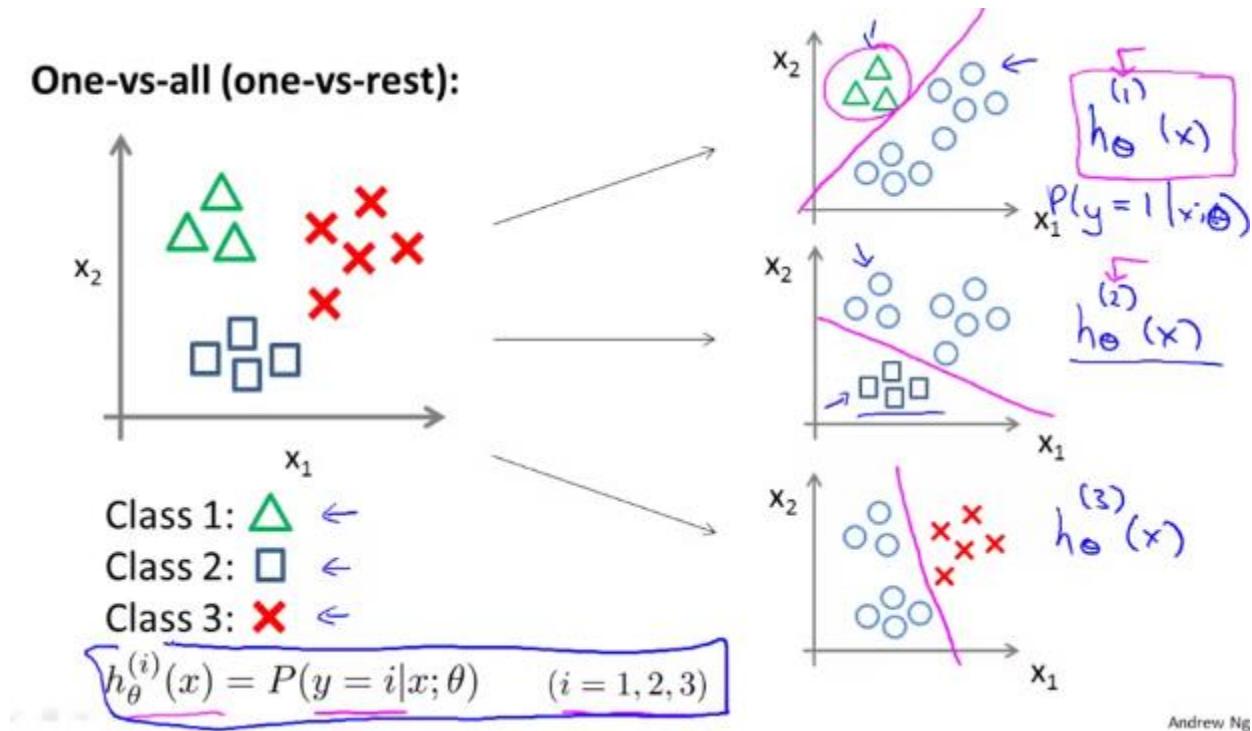


**Multi-class classification:**



So **the question is given the data set with three classes** where this is an example of one class, that's an example of a different class, and that's an example of yet a third class; **how do we get a learning algorithm to work for the setting?** We already know how to do **binary classification using logistic regression**, we know how we may **fit a straight line to separate the positive and negative classes**.

You see an idea called **one-vs-all classification**. We can then take this and **make it work for multi-class classification** as well. Here's how a one-vs-all classification works. And this is also sometimes called **one-vs-rest**.



Let's say we have a training set like that shown on the left, where we have three classes of  $y$  equals 1, we denote that with a triangle, if  $y$  equals 2, the square, and if  $y$  equals three, then the cross. What we're going to do is take our training set and **turn this into three separate binary classification problems**. I'll turn this into **three separate two class classification problems**.

So let's start with class one which is the triangle. We're gonna essentially create a new sort of fake training set where classes two and three get assigned to the negative class. And class one gets assigned to the positive class. You want to create a new training set like that shown on the right, and we're going to fit a classifier which I'm going to call  $h_{\theta}^1(x)$  where here the triangles are the positive examples and the circles are the negative examples. So think of the triangles being assigned the value of one and the circles assigned the value of zero. And we're just going to train a standard logistic regression classifier and maybe that will give us a position boundary that looks like that. Okay? This **superscript one** here stands for **class one**, so we're doing this for the triangles of class one. Next we do the same thing for class two. Gonna take the squares and assign the squares as the positive class, and assign everything else, the triangles and the crosses, as a negative class. And then we fit a second logistic regression classifier and call this  $h_{\theta}^2(x)$ , where the superscript two denotes that we're now doing this, treating the square class as the positive class. And maybe we get classifier like that. And finally, we do the same thing for the third class and fit a third classifier  $h_{\theta}^3(x)$ , and maybe this will give us a decision boundary, give us a classifier. This separates the positive and negative examples like that.

So to summarize, what we've done is, we've fit three classifiers. So, for  $i = 1, 2, 3$ , we'll fit a classifier  $h_{\theta}^i(x)$ . Thus trying to estimate **what is the probability that  $y$  is equal to class  $i$** , given  $x$  and parameterized by  $\theta$ . Right? So in the first instance for this first one up here, this classifier was learning to recognize the triangles. So it's thinking of the triangles as a positive

class, so  $h$  of superscript one is essentially trying to estimate what is the probability that the  $y$  is equal to one, given that  $x$  is parameterized by theta. And similarly, this is treating the square class as a positive class and so it's trying to estimate the probability that  $y = 2$  and so on.

**So we now have three classifiers, each of which was trained to recognize one of the three classes.**

### One-vs-all

Train a logistic regression classifier  $h_{\theta}^{(i)}(x)$  for each class  $i$  to predict the probability that  $y = i$ .

On a new input  $x$ , to make a prediction, pick the class  $i$  that maximizes

$$\max_i h_{\theta}^{(i)}(x)$$

Just to summarize, what we've done is we want to train a logistic regression classifier  $h$  superscript  $i$  of  $x$  for each class  $i$  to predict the probability that  $y$  is equal to  $i$ .

Finally to make a prediction, when we're given a new input  $x$ , and we want to make a prediction. **What we do is we just run all three of our classifiers on the input  $x$  and we then pick the class  $i$  that maximizes the three.** So we just basically pick the classifier, I think whichever one of the three classifiers is most confident and so the most enthusiastically says that it thinks it has the right class. So **whichever value of  $i$  gives us the highest probability** we then predict  $y$  to be that value.

So that's it for multi-class classification and **one-vs-all method**. And with this little method you can now take the logistic regression classifier and make it work on multi-class classification problems as well.

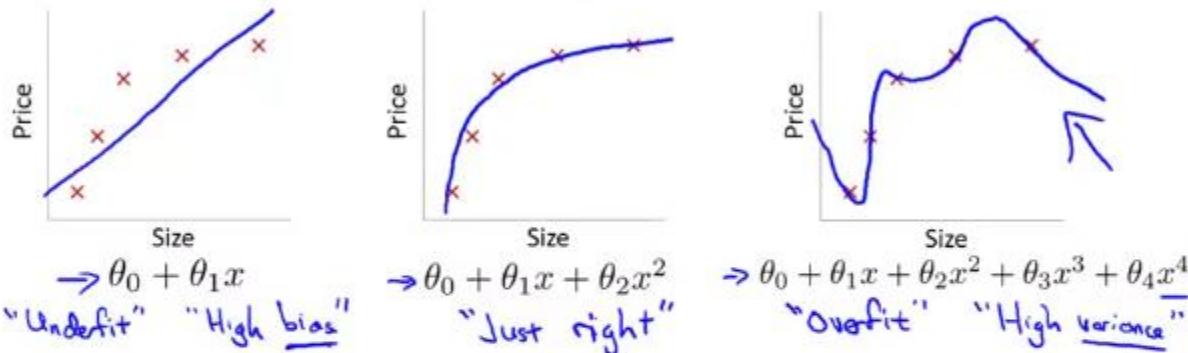
## Solving the Problem of Overfitting

### The Problem of Overfitting

By now, you've seen a couple different learning algorithms, **linear regression** and **logistic regression**. They work well for many problems, but when you apply them to certain machine learning applications, they can run into **a problem called overfitting that can cause them to perform very poorly**. What I'd like to do in this video is explain to you what is this overfitting problem, and in the next few videos after this, we'll talk about **a technique called regularization, that will allow us to ameliorate or to reduce this overfitting problem** and get these learning algorithms to maybe work much better.

**Underfitting / Overfitting: High Bias and High Variance**

So what is overfitting? Let's keep using our running example of predicting housing prices with linear regression where we want to predict the price as a function of the size of the house.

**Example: Linear regression (housing prices)**

**Overfitting:** If we have too many features, the learned hypothesis may fit the training set very well ( $J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 \approx 0$ ), but fail to generalize to new examples (predict prices on new examples).

One thing we could do is fit a linear function to this data, and if we do that, maybe we get that sort of straight line fit to the data. But this isn't a very good model. Looking at the data, it seems pretty clear that as the size of the housing increases, the housing prices plateau, or kind of flattens out as we move to the right and so this algorithm does not fit the training set very well and we call this problem **underfitting**, and another term for this is that **this algorithm has high bias**. Both of these roughly mean that it's just not even fitting the training data very well. The term bias is kind of a historical or technical one, but the idea is that if a fitting a straight line to the data, then, it's as if the **algorithm has a very strong preconception, or a very strong bias** that **housing prices** are going to vary **linearly** with their **size** and despite the data to the contrary, despite the evidence to the contrary its preconceptions still are **biased**, still causes it to fit a straight line and this ends up being a poor fit to the data.

Now, in the middle, we could fit a **quadratic function** to the data and with this data set, we fit the quadratic function, maybe, we get that kind of curve and that **works pretty well**.

And, at the other extreme, would be, if we were to fit, say a fourth other polynomial to the data. So, here we have five parameters, theta zero through theta four, and, with that, we can actually fit a curve that passes through all five of our training examples. You might get a curve that looks like this. That, on the one hand, seems to do a very good job fitting the training set and, that it passes through all of my data, at least. But, this is still a very wiggly curve, right? So, it's going up and down all over the place, and, we don't actually think that's such a good model for predicting housing prices. So, this problem we call **overfitting**, and, another term for this is that **this algorithm has high variance**. The term high variance is another sort of historical or technical one. But, the intuition is that, if we're fitting such a **high order polynomial**, then, the hypothesis

can fit, you know, it's almost as if it can fit almost any function and this face of possible hypothesis is just too large, it's **too variable**. And **we don't have enough data to constrain it to give us a good hypothesis so that's called overfitting**.

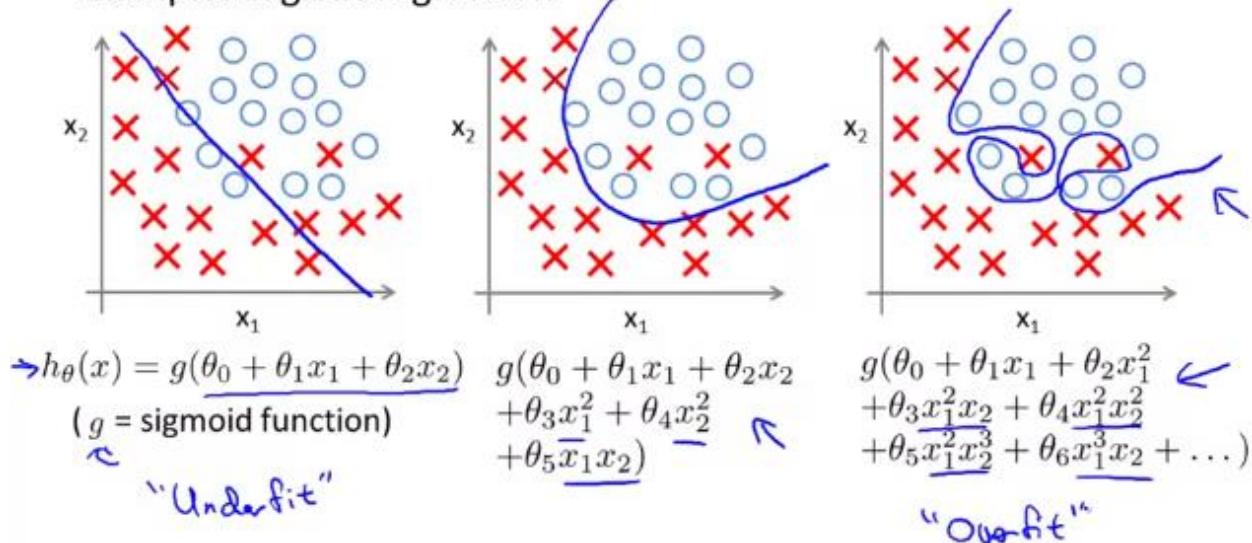
And in the middle, there isn't really a name but I'm just going to write, you know, "**Just right**", where **a second degree polynomial**, quadratic function seems to be **just right** for fitting this data.

To recap a bit, the problem of **over fitting** comes when if we have **too many features**, then the learning hypothesis may fit the training set very well. So, your **cost function** may actually be **very close to zero or may be even zero exactly**, but you may then end up with a curve like this that, you know tries too hard to fit the training set, so that it even **fails to generalize to new examples** and fails to predict prices on new examples well, and here the term generalized refers to how well a hypothesis applies even to new examples. That is to data to houses that it has not seen in the training set.

On this slide, we looked at **over fitting** for the case of **linear regression**. A similar thing can apply to **logistic regression** as well.

Here is a **logistic regression** example with two features  $x_1$  and  $x_2$ .

### Example: Logistic regression



One thing we could do, is fit logistic regression with just a simple hypothesis like this, where, as usual,  $g$  is my **sigmoid function**. And if you do that, you end up with a hypothesis, trying to use, maybe, just a straight line to separate the positive and the negative examples. And this doesn't look like a very good fit to the hypothesis. So, once again, this is an example of **underfitting** or of the hypothesis having **high bias**.

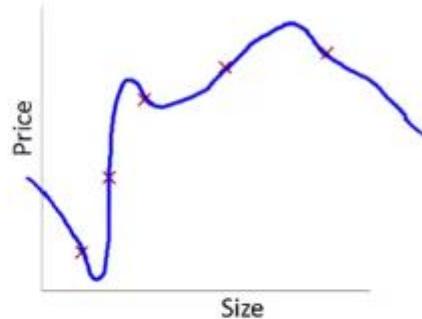
In contrast, if you were to add to your features these **quadratic terms**, then, you could get a decision boundary that might look more like this. And, you know, that's a **pretty good fit** to the data. Probably, about as good as we could get, on this training set.

And, finally, at the other extreme, if you were to fit a very **high-order polynomial**, if you were to generate lots of high-order polynomial terms of speeches, then, logistical regression may contort itself, may try really hard to find a decision boundary that fits your training data or go to great lengths to contort itself to fit every single training example well. And, you know, if the features  $x_1$  and  $x_2$  offer predicting, maybe, the cancer to the, you know, cancer is a malignant, benign breast tumors, this doesn't, this really doesn't look like a very good hypothesis, for making predictions. And so, once again, this is an instance of overfitting and, of a hypothesis having **high variance** and not really, and, being unlikely to generalize well to new examples.

Later, in this course, when we talk about debugging and diagnosing things that can go wrong with learning algorithms, we'll give you specific tools to recognize when overfitting and, also, when underfitting may be occurring. But, for now, let's talk about the problem of, if we think overfitting is occurring, what can we do to address it? In the previous examples, we had one or two dimensional data so, we could just plot the hypothesis and see what was going on and select the appropriate degree polynomial. So, earlier for the housing prices example, we could just plot the hypothesis and, you know, maybe see that it was fitting the sort of very wiggly function that goes all over the place to predict housing prices. And we could then use figures like these to select an appropriate degree polynomial. So **plotting the hypothesis**, could be **one way** to try to **decide what degree polynomial** to use.

### Addressing overfitting:

- $x_1$  = size of house
- $x_2$  = no. of bedrooms
- $x_3$  = no. of floors
- $x_4$  = age of house
- $x_5$  = average income in neighborhood
- $x_6$  = kitchen size
- :
- $x_{100}$



But that doesn't always work. And, in fact more often we may have learning problems that where we just have a lot of features. And there is not just a matter of selecting what degree polynomial. And, in fact, when we have so many features, it also becomes much harder to plot the data and it becomes much **harder to visualize it, to decide what features to keep or not**. So concretely, if we're trying predict housing prices sometimes we can just have a lot of different features. And all of these features seem, you know, maybe they seem kind of useful. But, if we have a **lot of features**, and, very little training data, then, **over fitting** can become a problem.

In order to address over fitting, there are **two main options** for things that we can do.

- The **first option** is, to try to **reduce the number of features**.

## Addressing overfitting:

### Options:

1. Reduce number of features.
  - — Manually select which features to keep.
  - — Model selection algorithm (later in course).

Concretely, one thing we could do is manually look through the list of features, and, use that to try to decide, which are the more important features, and, therefore, which are the features we should keep, and, which are the features we should throw out.

Later in this class, we will also talk about model selection algorithms. Which are algorithms for automatically deciding which features to keep and, which features to throw out. This idea of reducing the number of features can work well, and, can reduce over fitting. And, when we talk about model selection, we'll go into this in much greater depth. But, the disadvantage is that, by throwing away some of the features, is also throwing away some of the information you have about the problem. For example, maybe, all of those features are actually useful for predicting the price of a house, so, maybe, we don't actually want to throw some of our information or throw some of our features away.

- The second option, which we'll talk about in the next few videos, is regularization.
- 2. Regularization.
  - — Keep all the features, but reduce magnitude/values of parameters  $\theta_j$ .
  - Works well when we have a lot of features, each of which contributes a bit to predicting  $y$ .

Here, we're going to keep all the features, but we're going to reduce the magnitude or the values of the parameters  $\theta_j$ . And, this method works well, we'll see, when we have a lot of features, each of which contributes a little bit to predicting the value of  $y$ , like we saw in the housing price prediction example where we could have a lot of features, each of which are, you know, somewhat useful, so, maybe, we don't want to throw them away.

So, this subscribes the idea of regularization at a very high level. And, I realize that, all of these details probably don't make sense to you yet. But, in the next video, we'll start to formulate exactly how to apply regularization and, exactly what regularization means. And, then we'll start to figure out, how to use this, to make how learning algorithms work well and avoid overfitting.

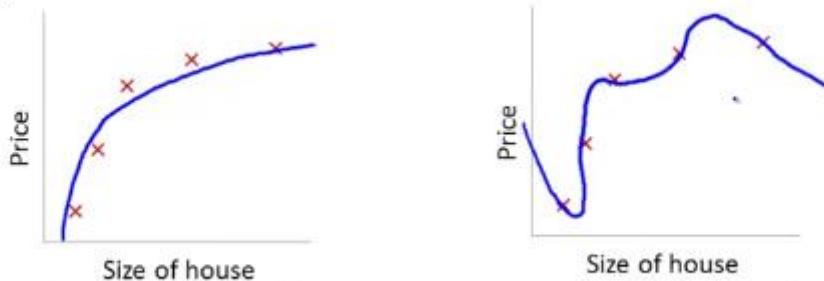
## Cost Function

In this video, I'd like to convey to you, the main intuitions behind how regularization works. And, we'll also write down the cost function that we'll use, when we were using regularization.

With the hand drawn examples that we have on these slides, I think I'll be able to convey part of the intuition. But, an even better way to see for yourself, how regularization works, is if you implement it, and see it work for yourself. And, if you do the appropriate exercises after this, you get the chance to see regularization in action for yourself. So, here is the intuition.

In the previous video, we saw that, if we were to fit a **quadratic function** to this data, it gives us a pretty **good fit** to the data. Whereas, if we were to fit an **overly high order degree polynomial**, we end up with a curve that may fit the training set very well, but, really not be a, but **overfit** the data poorly, and, not generalize well.

### Intuition

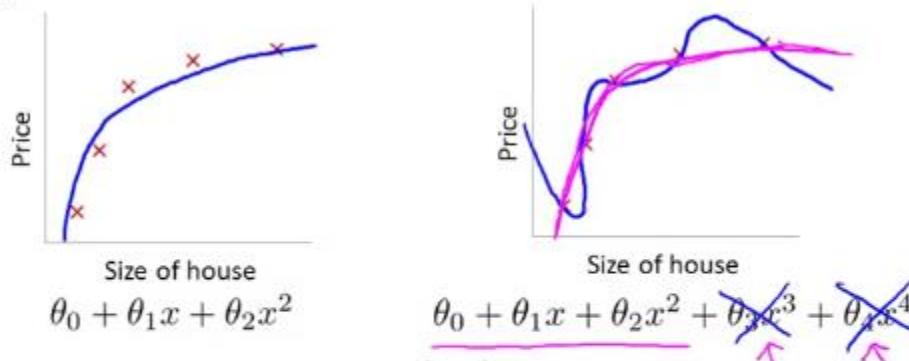


$$\theta_0 + \theta_1 x + \theta_2 x^2$$

$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

Consider the following, suppose we were to penalize, and, make the parameters theta 3 and theta 4 really small. Here's what I mean, here is our optimization objective, or here is our optimization problem, where we minimize our usual squared error cost function. Let's say I take this objective and modify it and add to it, plus **1000  $\theta_3^2$** , plus **1000  $\theta_4^2$** . 1000 I am just writing down as some huge number. Now, if we were to minimize this function, the only way to make this new cost function small is if  **$\theta_3$**  and  **$\theta_4$**  are small, right?

### Intuition



Suppose we penalize and make  $\theta_3, \theta_4$  really small.

$$\rightarrow \min_{\theta} \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + 1000 \underline{\theta_3^2} + 1000 \underline{\theta_4^2}$$

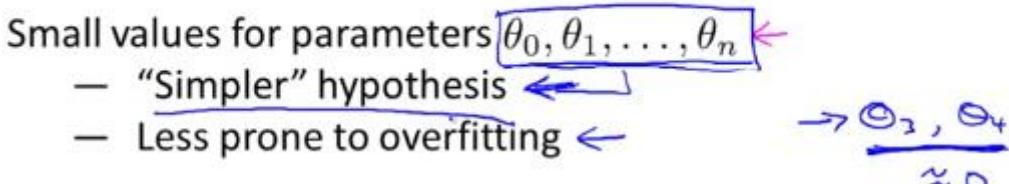
$\theta_3 \approx 0$      $\theta_4 \approx 0$

Because otherwise, if you have a 1000 times  **$\theta_3$** , this new cost function gonna be big. So when we minimize this new function we are going to end up with  **$\theta_3$**  close to 0 and  **$\theta_4$**  close to 0, as if we're getting rid of these two terms over there. And if we do that, well then, if  **$\theta_3$**  and  **$\theta_4$**  close to

0 then we are being left with a quadratic function, and, so, we end up with a fit to the data, that's, you know, **quadratic function plus** maybe, **tiny contributions from small terms,  $\theta_3, \theta_4$** , that they may be very close to 0. And, so, we end up with essentially, a quadratic function, which is good because this is a much better hypothesis.

In this particular example, we looked at the effect of **penalizing two of the parameter values being large**. More generally, **here is the idea behind regularization**.

### Regularization.



The idea is that, if we have **small values for the parameters**, then, having small values for the parameters will somehow, will usually correspond to having a **simpler hypothesis**. So, for our last example, we **penalize** just  $\theta_3$ , and  $\theta_4$ , and when both of these were close to zero, **we wound up with a much simpler hypothesis that was essentially a quadratic function**. But more broadly, if we penalize all the parameters usually that, we can think of that, as trying to give us a simpler hypothesis as well because when, you know, these parameters are close to 0 in this example that gave us a quadratic function. **But more generally, it is possible to show that having smaller values of the parameters corresponds to usually smoother functions as well for the simpler. And which are therefore, also, less prone to overfitting.** I realize that the reasoning for why having all the parameters be small, why that corresponds to a simpler hypothesis; I realize that reasoning may not be entirely clear to you right now. And it is kind of hard to explain unless you implement it yourself and see it for yourself. But I hope that the **example of having  $\theta_3$  and  $\theta_4$  be small and how that gave us a simpler hypothesis**, I hope that helps explain why, at least give some intuition as to why this might be true.

Let's look at the specific example.

## Regularization.

Small values for parameters  $\theta_0, \theta_1, \dots, \theta_n$

- "Simpler" hypothesis
- Less prone to overfitting

$$\rightarrow \boxed{\theta_3, \theta_4} \quad \uparrow \approx 0$$

Housing:

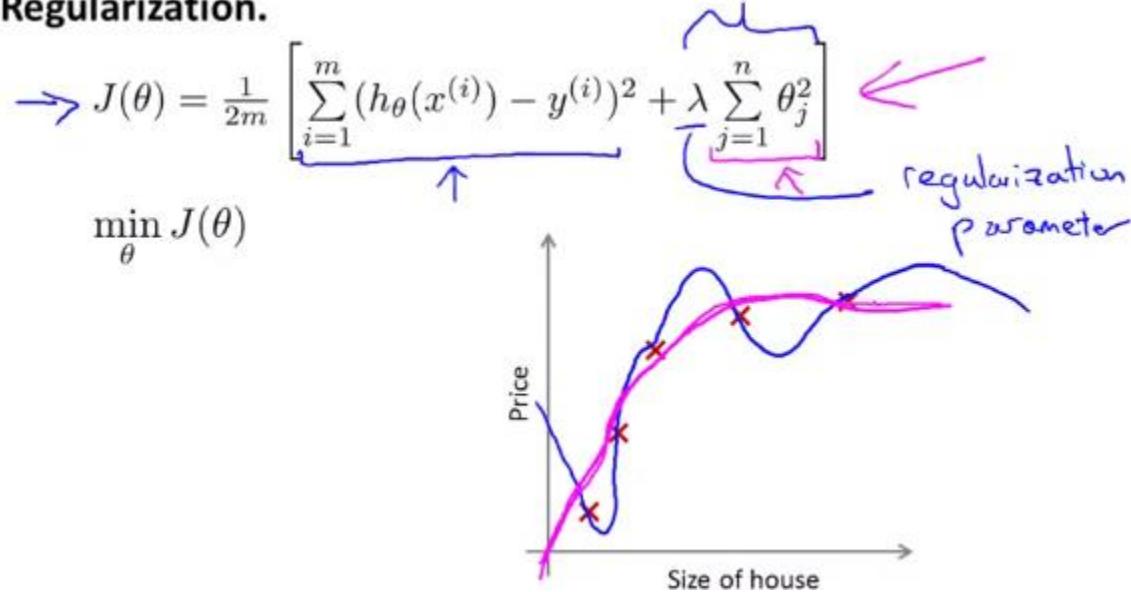
- Features:  $x_1, x_2, \dots, x_{100}$
- Parameters:  $\theta_0, \theta_1, \theta_2, \dots, \theta_{100}$

$$J(\theta) = \frac{1}{2m} \left[ \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^n \theta_j^2 \right]$$

~~$\theta_1, \theta_2, \theta_3, \dots, \theta_{100}$~~

For housing price prediction we may have our hundred features that we talked about where maybe  $x_1$  is the size,  $x_2$  is the number of bedrooms,  $x_3$  is the number of floors and so on. And we may we may have a hundred features. And unlike the polynomial example, we don't know, right, we don't know that  $\theta_3, \theta_4$  are the high order polynomial terms. So, if we have just a bag, if we have just a set of a hundred features, it's hard to pick in advance which are the ones that are less likely to be relevant. So we have a hundred or a hundred one parameters and we don't know which ones to pick, we don't know which parameters to pick, to try to shrink. So, in regularization, what we're going to do, is take our cost function, here is my cost function for linear regression, and what I'm going to do is, modify this cost function to shrink all of my parameters, because, you know, I don't know which one or two to try to shrink. So I am going to modify my cost function to add a term at the end. Like so we have square brackets here as well. When I add an extra regularization term at the end to shrink every single parameter and so this term we tend to shrink all of my parameters  $\theta_1, \theta_2, \theta_3$  up to  $\theta_{100}$ . By the way, by convention the summation here starts from one so I am not actually going penalize theta zero being large. That sort of the convention that, the sum i equals one through n, rather than i equals zero through n. But in practice, it makes very little difference, and, whether you include, you know,  $\theta_0$  or not, in practice, make very little difference to the results. But by convention, usually, we regularize only  $\theta_1$  through  $\theta_{100}$ .

Writing down our regularized optimization objective, our regularized cost function again. Here it is.

**Regularization.**

Here's  $J(\theta)$  where, this term on the right is a **regularization term** and **lambda** here is called the **regularization parameter** and what lambda does, is it controls a trade off between two different goals.

- The **first goal**, captured by the **first term**, the objective, is that we would like to train, is that we would like to fit the training data well. We would like to fit the training set well.
- And the **second goal** is, we want to **keep the parameters small**, and that's captured by the **second term**, by the **regularization objective**, and by the regularization term. And what lambda, the regularization parameter does is it controls the trade off between these two goals, between the goal of fitting the training set well and the goal of keeping the parameter small and therefore keeping the hypothesis relatively simple to avoid overfitting.

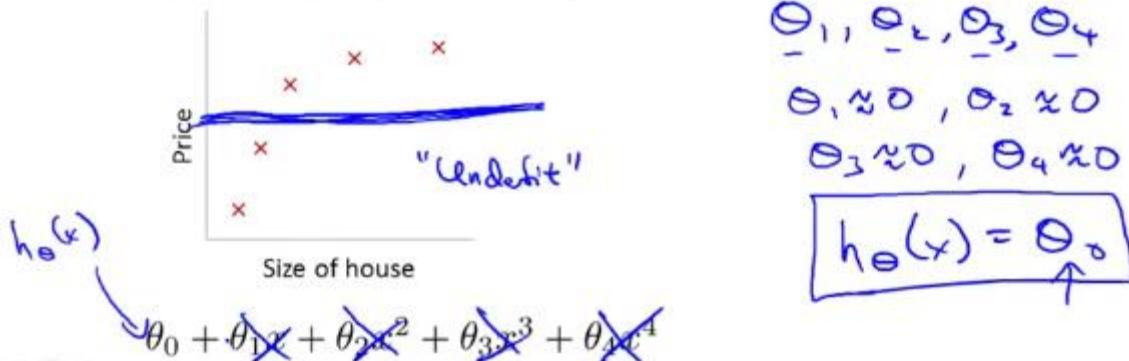
For our housing price prediction example, whereas, previously, if we had fit a very high order polynomial, we may have wound up with a very, sort of wiggly or curvy function like this. If you still fit a high order polynomial with all the polynomial features in there, but instead, you just make sure, to use this sole of regularized objective, then what you can get out is in fact a curve that isn't quite a quadratic function, but is much smoother and much simpler and maybe a curve like the magenta line that, you know, gives a much better hypothesis for this data.

Once again, I realize it can be a bit difficult to see why strengthening the parameters can have this effect, but if you implement these algorithms yourselves with regularization you will be able to see this effect firsthand.

In regularized linear regression, we choose  $\theta$  to minimize

$$J(\theta) = \frac{1}{2m} \left[ \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^n \theta_j^2 \right]$$

What if  $\lambda$  is set to an extremely large value (perhaps for too large for our problem, say  $\lambda = 10^{10}$ )?



In regularized linear regression, if the regularization parameter lambda is set to be very large, then what will happen is that we will end up penalizing the parameters  $\theta_1, \theta_2, \theta_3, \theta_4$  very highly. That is, if our hypothesis is this one, down at the bottom, and if we end up penalizing  $\theta_1, \theta_2, \theta_3, \theta_4$  very heavily, then we end up with all of these parameters close to zero, right?  $\theta_1$  will be close to zero;  $\theta_2$  will be close to zero.  $\theta_3$  and  $\theta_4$  will end up being close to zero. And if we do that, it's as if we're getting rid of these terms in the hypothesis, so that we're just left with a hypothesis that will say that. It says that, well, housing prices are equal to  $\theta_0$ , and that is akin to fitting a flat horizontal straight line to the data. And this is an example of underfitting, and in particular this hypothesis, this straight line it just fails to fit the training set well. It's just a flat straight line, it doesn't go, you know, go near. It doesn't go anywhere near most of the training examples. And another way of saying this is that this hypothesis has too strong a preconception or too high bias that housing prices are just equal to  $\theta_0$ , and despite the clear data to the contrary, you know chooses to fit a sort of, flat line, just a flat horizontal line. I didn't draw that very well. This just a horizontal flat line to the data.

So for regularization to work well, some care should be taken, to choose a good choice for the regularization parameter lambda as well. And when we talk about multi-selection later in this course, we'll talk about a way, a variety of ways for automatically choosing the regularization parameter lambda as well.

So, that's the idea of the high regularization and the cost function reviews in order to use regularization. In the next two videos, let's take these ideas and apply them to linear regression and to logistic regression, so that we can then get them to avoid overfitting.

## Regularized Linear Regression

For **linear regression**, we have previously worked out two learning algorithms. One based on **gradient descent** and one based on the **normal equation**.

In this video, we'll take those two algorithms and generalize them to the case of **regularized linear regression**.

Here's the optimization objective that we came up with last time for regularized linear regression. This **first part** is our **usual objective for linear regression**. And we now have this **additional regularization term**, where **lambda is our regularization parameter**, and we like to find parameters  $\theta$  that minimizes this cost function, this regularized cost function,  $J(\theta)$ .

### Regularized linear regression

$$J(\theta) = \frac{1}{2m} \left[ \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \underbrace{\sum_{j=1}^n \theta_j^2}_{\text{regularization term}} \right]$$

$$\min_{\theta} J(\theta)$$

Previously, we were using **gradient descent for the original cost function without the regularization term**. And we had the following algorithm, for regular linear regression, without regularization, we would repeatedly update the parameters  $\theta_j$  as follows for  $j$  equals 0, 1, 2, up through n.

### Gradient descent

Repeat {

$$\rightarrow \theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \quad (j = 0, 1, 2, 3, \dots, n)$$

Let me take this and just write the case for  $\theta_0$  separately. So I'm just going to write the update for  $\theta_0$  separately than for the update for the parameters 1, 2, 3, and so on up to n. And so this is, I haven't changed anything yet, right.

**Gradient descent**

$$\frac{\partial}{\partial} \theta_0, \theta_1, \dots, \theta_n$$

Repeat {

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

$$\rightarrow \theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \quad (j = 1, 2, 3, \dots, n)$$

}

This is just writing the update for  $\theta_0$  separately from the updates for  $\theta_1, \theta_2, \theta_3$ , up to  $\theta_n$ . And the reason I want to do this is you may remember that for our regularized linear regression, we penalize the parameters  $\theta_1, \theta_2$ , and so on up to  $\theta_n$ . But we don't penalize  $\theta_0$ . So, when we modify this algorithm for regularized linear regression, we're going to end up treating  $\theta_0$  slightly differently.

**Gradient descent**

$$\frac{\partial}{\partial} \theta_0, \theta_1, \dots, \theta_n$$

Repeat {

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

$$\begin{aligned} \rightarrow \theta_j &:= \theta_j - \alpha \left[ \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \theta_j \right] \\ &\quad (j = 1, 2, 3, \dots, n) \\ \rightarrow \theta_j &:= \theta_j \left( 1 - \alpha \frac{\lambda}{m} \right) - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \end{aligned}$$

$$\Rightarrow J(\theta)$$

$$\theta_j$$

$$1 - \alpha \frac{\lambda}{m} < 1$$

$$0.99$$

$$\theta_j \times 0.99$$

Concretely, if we want to take this algorithm and modify it to use the regularise objective, all we need to do is take this term at the bottom and modify it as follows. We'll take this term and add minus lambda over m times  $\theta_j$ . And if you implement this, then you have gradient descent for trying to minimize the regularized cost function  $J(\theta)$ . And concretely, I'm not gonna do the calculus to prove it, but concretely if you look at this term, this term that I've written in square brackets, if you know calculus it's possible to prove that that term is the partial derivative with respect to  $J(\theta)$  using the new definition of  $J(\theta)$  with the regularization term. And similarly, this term up on top which I'm drawing the cyan box, that's still the partial derivative with respect of  $\theta_0$  of  $J(\theta)$ .

If you look at the update for  $\theta_j$ , it's possible to show something very interesting.

Concretely,  $\theta_j$  gets updated as  $\theta_j$  minus  $\alpha$  times and then you have this other term here that depends on  $\theta_j$ . So if you group all the terms together that depend on  $\theta_j$ , you can show that this update can be written equivalently as follows. And all I did was add  $\theta_j$  here is, so  $\theta_j$  times 1 and this term is, right,  $\lambda$  over m, there's also an  $\alpha$  here, so you end up with  $\alpha \lambda$  over m multiplied into  $\theta_j$ . And this term here, 1 minus  $\alpha$  times  $\lambda$  m, is a pretty interesting term. It has a pretty interesting effect.

Concretely, this term, **1 minus alpha times lambda over m** is going to be a number that is usually it's a number that **is usually a little bit less than one**, because **alpha times lambda over m is going to be positive**, and usually **if your learning rate is small and if m is large, this is** usually pretty small. So this term here is gonna be a number that's usually a little bit less than 1, so think of it as a number like **0.99**, let's say. And so the effect of our update to  $\theta_j$  is, we're going to say that  $\theta_j$  gets replaced by  $\theta_j$  times 0.99, right? So  $\theta_j$  times 0.99 has the effect of shrinking  $\theta_j$  a little bit towards zero. So this makes  $\theta_j$  a bit smaller. And more formally, this makes the square norm of  $\theta_j$  a little bit smaller. And then after that, the second term here, that's actually exactly the same as the original gradient descent update that we had, before we added all this regularization stuff.

So, hopefully this gradient descent, hopefully this update makes sense. When we're using a regularized linear regression and what we're doing is **on every iteration we're multiplying theta j by a number that's a little bit less than one**, so its shrinking the parameter a little bit, and then we're performing a similar update as before. Of course that's just the intuition behind what this particular update is doing. Mathematically what it's doing is it's exactly gradient descent on the cost function  $J(\theta)$  that we defined on the previous slide that uses the regularization term.

Gradient descent was just one of our two algorithms for fitting a linear regression model.

**The second algorithm** was the one based on the **normal equation**, where what we did was we created the **design matrix X** where **each row corresponded to a separate training example**. And we created a **vector y**, so this is a vector, that's an m dimensional vector. And that contained the labels from my training set. So whereas X is an m by **(n+1) dimensional matrix**, y is an **m dimensional vector**. And in order **to minimize the cost function J**, we found that one way to do so is to **set theta to be equal to this**. Right, you have **X transpose X, inverse, X transpose y**. I'm leaving room here to fill in stuff of course.

**Normal equation**

$$X = \begin{bmatrix} (x^{(1)})^T \\ \vdots \\ (x^{(m)})^T \end{bmatrix} \leftarrow \text{m} \times (n+1)$$

$$\rightarrow \min_{\theta} J(\theta) \quad \frac{\partial}{\partial \theta_j} J(\theta) \stackrel{\text{set}}{=} 0 \quad \Rightarrow \quad \begin{pmatrix} X^T X + \lambda I & X^T y \end{pmatrix}^{-1}$$

And what this value for  $\theta$  does is this minimizes the cost function  $J(\theta)$ , when we were not using regularization. Now that we are using regularization, if you were to derive what the minimum is, and just to give you a sense of how to derive the minimum, the way you derive it is you take partial derivatives with respect to each parameter. Set this to zero, and then do a bunch of math and you can then show that it's a formula like this that minimizes the cost function.

And concretely, if you are using regularization, then this formula changes as follows.

**Normal equation**

$$X = \begin{bmatrix} (x^{(1)})^T \\ \vdots \\ (x^{(m)})^T \end{bmatrix} \leftarrow \text{m} \times (n+1)$$

$$\rightarrow \min_{\theta} J(\theta) \quad \frac{\partial}{\partial \theta_j} J(\theta) \stackrel{\text{set}}{=} 0 \quad \Rightarrow \quad \begin{pmatrix} X^T X + \lambda I & X^T y \end{pmatrix}^{-1}$$

e.g. n=2  $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (n+1) \times (n+1)$

Inside this parenthesis, you end up with a matrix like this. 0, 1, 1, 1, and so on, 1, until the bottom. So this thing over here is a matrix whose upper left-most entry is 0. There are ones on the diagonals, and then zeros everywhere else in this matrix. Because I'm drawing this rather sloppily. But as a concrete example, if  $n = 2$ , then this matrix is going to be a three by three matrix. More generally, this matrix is an  $(n+1)$  by  $(n+1)$  dimensional matrix. So if  $n = 2$ , then that matrix becomes something that looks like this. It would be 0, and then 1s on the diagonals, and then 0s on the rest of the diagonals. And once again, I'm not going to show this derivation, which is frankly somewhat long and involved, but it is possible to prove that if you are using the

new definition of  $J(\theta)$ , with the regularization objective, then **this new formula for  $\theta$  is the one that will give you, the global minimum of  $J(\theta)$ .**

So finally I want to just quickly describe the issue of **non-invertibility**.

This is relatively advanced material, so you should consider this as optional. And feel free to skip it, or if you listen to it and possibly it doesn't really make sense, don't worry about it either. But earlier when I talked about the normal equation method, we also had an optional video on the non-invertibility issue. So this is another optional part to this, sort of an add-on to that earlier optional video on non-invertibility.

Now, consider a setting where **m, the number of examples, is less than or equal to n, the number of features.** If you have fewer examples than features, then this matrix,  $X$  transpose  $X$  will be **non-invertible, or singular**. Or the other term for this is the matrix will be **degenerate**. And if you implement this in Octave anyway and you use the **pinv** function to take the pseudo inverse, it will kind of do the right thing, but it's **not clear that it would give you a very good hypothesis**, even though numerically the Octave pinv function will give you a result that kinda makes sense. But if you were doing this in a different language, and if you were taking just the regular inverse, which in Octave denoted with the function inv, we're trying to take the regular inverse of  $X$  transpose  $X$ . Then in this setting, you find that  $X$  transpose  $X$  is singular, is non-invertible, and if you're doing this in different program language and using some linear algebra library to try to take the inverse of this matrix, it **just might not work because that matrix is non-invertible or singular**. **Fortunately, regularization also takes care of this for us.** And concretely, so long as the regularization parameter **lambda is strictly greater than 0**, it is actually possible to prove that this matrix,  $X$  transpose  $X$  plus lambda times this funny matrix here, it is possible to prove that this matrix will not be singular and that this **matrix will be invertible**.

**So using regularization also takes care of any non-invertibility issues of the  $X$  transpose  $X$  matrix as well.**

So you now know how to **implement regularized linear regression**. Using this you'll be able to **avoid overfitting even if you have lots of features in a relatively small training set**. And this should let you get linear regression to work much better for many problems.

In the next video we'll take this regularization idea and apply it to logistic regression. So that you'd be able to get **logistic regression to avoid overfitting** and perform much better as well.

## Summary - Regularized Linear Regression

**Note:** [8:43 - It is said that  $X$  is non-invertible if  $m \leq n$ . The correct statement should be that  $X$  is non-invertible if  $m < n$ , and may be non-invertible if  $m = n$ .

We can apply regularization to both linear regression and logistic regression. We will approach linear regression first.

## Gradient Descent

We will modify our gradient descent function to separate out  $\theta_0$  from the rest of the parameters because we do not want to penalize  $\theta_0$ .

$$\begin{aligned}
 & \text{Repeat} \{ \\
 & \quad \theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)} \\
 & \quad \theta_j := \theta_j - \alpha \left[ \left( \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \right] \quad j \in \{1, 2, \dots, n\} \\
 & \}
 \end{aligned}$$

The term  $\lambda/m \theta_j$  performs our **regularization**. With some manipulation our update rule can also be represented as:

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

The first term in the above equation,  $1 - \alpha \lambda/m$  will always be less than 1. Intuitively you can see it as reducing the value of  $\theta_j$  by some amount on every update. Notice that the second term is now exactly the same as it was before.

## Normal Equation

Now let's approach regularization using the alternate method of the non-iterative normal equation.

To add in regularization, the equation is the same as our original, except that we add another term inside the parentheses:

$$\theta = (X^T X + \lambda \cdot L)^{-1} X^T y$$

where  $L = \begin{bmatrix} 0 & & & \\ & 1 & & \\ & & 1 & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix}$

$L$  is a matrix with 0 at the top left and 1's down the diagonal, with 0's everywhere else. It should have dimension  $(n+1) \times (n+1)$ . Intuitively, this is the identity matrix (though we are not including  $x_0$ ), multiplied with a single real number  $\lambda$ .

Recall that if  $m < n$ , then  $X^T X$  is non-invertible. However, when we add the term  $\lambda \cdot L$ , then  $X^T X + \lambda \cdot L$  becomes invertible.

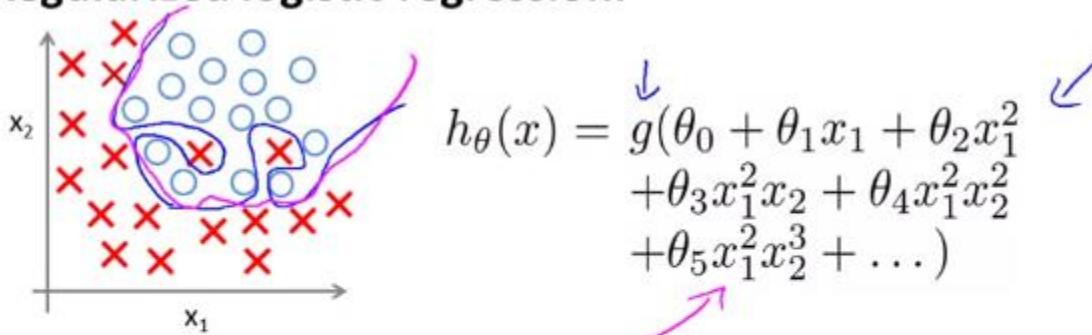
## Regularized Logistic Regression

For logistic regression, we previously talked about two types of optimization algorithms. We talked about how to use **gradient descent to optimize as cost function  $J(\theta)$**  and we also talked about **advanced optimization methods**. Ones that require that you provide a way to compute your cost function  $J(\theta)$  and that you provide a way to compute the derivatives.

In this video, we'll show how you can adapt both of those techniques, **both gradient descent and the more advanced optimization techniques** in order to have them work for **regularized logistic regression**. So, here's the idea.

We saw earlier that Logistic Regression can also be **prone to overfitting** if you fit it with a very, sort of, high order **polynomial features** like this, where  $g$  is the **sigmoid function** and in particular you may end up with a hypothesis, you know, whose decision boundary is this sort of an overly complex and extremely contortive function that really isn't such a great hypothesis for this training set, and more generally **if you have logistic regression with a lot of features, not necessarily polynomial ones**, but just with a lot of features **you can end up with overfitting**.

### Regularized logistic regression.



Cost function:

$$\Rightarrow J(\theta) = - \left[ \frac{1}{m} \sum_{i=1}^m y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_{\theta}(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

$\boxed{\theta_1, \theta_2, \dots, \theta_n}$

This was our cost function for logistic regression. And if we want to modify it to use regularization, all we need to do is add to it the following term plus lambda over 2m, sum from j equals 1, and as usual sum from j equals 1 rather than the sum from j equals 0, of  $\theta_j$  squared, and this has the effect therefore of penalizing the parameters  $\theta_1 \theta_2$  and so on up to  $\theta_n$  from being too large. And if you do this, then it will have the effect that even though you're fitting a very high order polynomial with a lot of parameters, so long as you apply regularization and keep the parameters small you're more likely to get a decision boundary. You know, that maybe looks more like this. It looks more reasonable for separating the positive and the negative examples.

So, when using regularization even when you have a lot of features, the regularization can help take care of the overfitting problem. How do we actually implement this?

Well, for the original gradient descent algorithm, this was the update we had.

### Gradient descent

Repeat {

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

$$\rightarrow \theta_j := \theta_j - \alpha \left[ \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \theta_j \right] \quad (j = 1, 2, 3, \dots, n)$$

$\theta_1, \dots, \theta_n$

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

We will repeatedly perform the following update to  $\theta_j$ . This slide looks a lot like the previous one for linear regression. But what I'm going to do is write the update for  $\theta_0$  separately. So, the first line is for update for  $\theta_0$  and a second line is now my update for  $\theta_1$  up to  $\theta_n$ . Because I'm going to treat  $\theta_0$  separately. And in order to modify this algorithm, to use a regularized cost function, all I need to do is pretty similar to what we did for linear regression, is actually to just modify this second update rule as follows.

And, once again, this, you know, cosmetically looks identical what we had for linear regression. But of course is not the same algorithm as we had, because now the hypothesis is defined using this. So this is not the same algorithm as regularized linear regression. Because the hypothesis is different. Even though this update that I wrote down. It actually looks cosmetically the same as what we had earlier. We're working out gradient descent for regularized linear regression.

$$\rightarrow \theta_j := \theta_j - \alpha \left[ \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \theta_j \right] \leftarrow$$

(j = 1, 2, 3, ..., n)

$\theta_1, \dots, \theta_n$

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

And of course, just to wrap up this discussion, this term here in the square brackets, so this term here, this term is, of course, the new partial derivative for respect of  $\theta_j$  of the new cost function  $J(\theta)$  where  $J(\theta)$  here is the cost function we defined on a previous slide that does use regularization. So, that's gradient descent for regularized linear regression.

Let's talk about **how to get regularized linear regression to work using the more advanced optimization methods**. And just to remind you for those methods what we needed to do was to define the function that's called the cost function, that takes us input the parameter vector theta and once again in the equations we've been writing here we used 0 index vectors. So we had  $\theta_0$  up to  $\theta_n$ . But because Octave indexes the vectors starting from 1.  $\theta_0$  is written in Octave as  $\theta_1$ ,  $\theta_1$  is written in Octave as  $\theta_2$ , and so on down to  $\theta_{n+1}$ . And what we needed to do was provide a function.

Let's provide a function called **costFunction** that we would then pass in to what we have, what we saw earlier, we will use the fminunc and then you know @ **costFunction**, and so on, right. But the F min, u n c was the **F min unconstrained** and this will work with **fminunc** was what **will take the cost function and minimize it for us**. So the two main things that the cost function needed to return were first **jVal**. And for that, we need to write code to compute the cost function  $J(\theta)$ .

### Advanced optimization

```

→ function [jVal, gradient] = costFunction(theta)
    jVal = [code to compute J(θ)];
    → 
$$J(\theta) = \left[ -\frac{1}{m} \sum_{i=1}^m y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

    → gradient(1) = [code to compute  $\frac{\partial}{\partial \theta_0} J(\theta)$ ];
        
$$\frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)}$$

    → gradient(2) = [code to compute  $\frac{\partial}{\partial \theta_1} J(\theta)$ ];
        
$$\left( \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_1^{(i)} \right) + \frac{\lambda}{m} \theta_1$$

    → gradient(3) = [code to compute  $\frac{\partial}{\partial \theta_2} J(\theta)$ ];
        
$$\vdots \quad \left( \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_2^{(i)} \right) + \frac{\lambda}{m} \theta_2$$

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

```

Now, when we're using regularized logistic regression, of course the cost function  $J(\theta)$  changes and, in particular, now a **cost function needs to include this additional regularization term at the end as well**. So, when you compute  $J(\theta)$  be sure to include that term at the end. And then, the other thing that this cost function thing needs to derive was a gradient. So gradient one needs to be set to the partial derivative of  $J(\theta)$  with respect to  $\theta_0$ , gradient two needs to be set to that, and so on. Once again, the index is off by one. Right, because of the indexing from one that Octave uses. And looking at these terms, this term over here, we actually worked this out on a previous slide, is actually equal to this. It doesn't change. Because the derivative for  $\theta_0$  doesn't change compared to the version without regularization. And the other terms do change. And in particular the derivative with respect to  $\theta_1$ , we worked this out on the previous slide as well, is equal to, you know, the original term and then minus lambda over m times  $\theta_1$ . Just so we make sure we pass this correctly, if we can add parentheses here. Right, so the summation doesn't extend. And

similarly, you know, this other term here looks like this, with this additional term that we had on the previous slide that corresponds to the gradient from their regularization objective. So if you implement this cost function and pass this into **fminunc** or to one of those advanced optimization techniques, that will minimize the new regularized cost function  $J(\theta)$ . And the parameters you get out will be the ones that correspond to logistic regression with regularization. So, now you know how to implement regularized logistic regression.

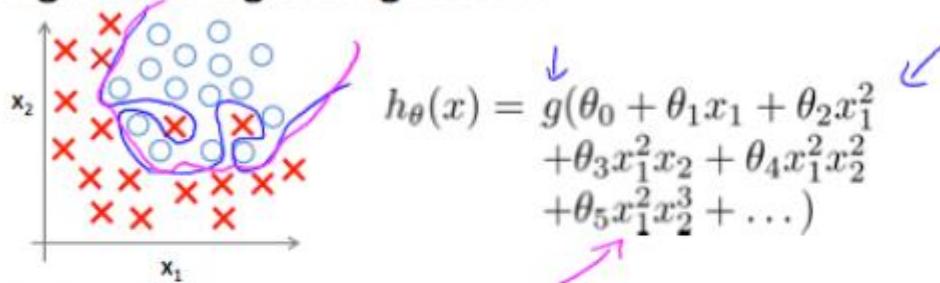
When I walk around Silicon Valley, I live here in Silicon Valley, there are a lot of engineers that are frankly, making a ton of money for their companies using machine learning algorithms. And I know we've only been, you know, studying this stuff for a little while. But if you understand linear regression, logistic regression, the advanced optimization algorithms and regularization, by now, frankly, you probably know quite a lot more machine learning than many, certainly not all, but you probably know quite a lot more machine learning right now than frankly, many of the Silicon Valley engineers out there having very successful careers you know, making tons of money for the companies or building products using machine learning algorithms.

So, congratulations. You've actually come a long ways. And you can actually, you actually know enough to apply this stuff and get to work for many problems. So congratulations for that. But of course, there's still a lot more that we want to teach you, and in the next set of videos after this, we'll start to talk about a very powerful class of non-linear classifier. So whereas linear regression, logistic regression, you know, you can form polynomial terms, but it turns out that there are much more powerful nonlinear classifiers, that can, than sort of polynomial regression. And in the next set of videos after this one, I'll start telling you about them. So that you have even more powerful learning algorithms than you have now to apply to different problems.

## Summary - Regularized Logistic Regression

We can regularize logistic regression in a similar way that we regularize linear regression. As a result, we can avoid overfitting. The following image shows how the regularized function, displayed by the pink line, is less likely to overfit than the non-regularized function represented by the blue line:

## Regularized logistic regression.



**Cost function:**

$$\Rightarrow J(\theta) = - \left[ \frac{1}{m} \sum_{i=1}^m y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

$\theta_1, \theta_2, \dots, \theta_n$

### Cost Function

Recall that our cost function for logistic regression was:

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

We can regularize this equation by adding a term to the end:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

The second sum,  $\sum_{j=1}^n \theta_j^2$  means to explicitly exclude the **bias term**,  $\theta_0$ . I.e. the  $\theta$  vector is indexed from 0 to n (holding n+1 values,  $\theta_0$  through  $\theta_n$ ), and this sum explicitly skips  $\theta_0$ , by running from 1 to n, skipping 0. Thus, when computing the equation, we should continuously update the two following equations:

### Gradient descent

Repeat {

$$\begin{aligned} \rightarrow \quad \theta_0 &:= \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)} \\ \rightarrow \quad \theta_j &:= \theta_j - \alpha \underbrace{\left[ \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \theta_j \right]}_{\substack{(j=1, 2, 3, \dots, n) \\ \theta_1, \dots, \theta_n}} \leftarrow \\ &\quad \} \end{aligned}$$

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

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

# Neural Networks: Representation

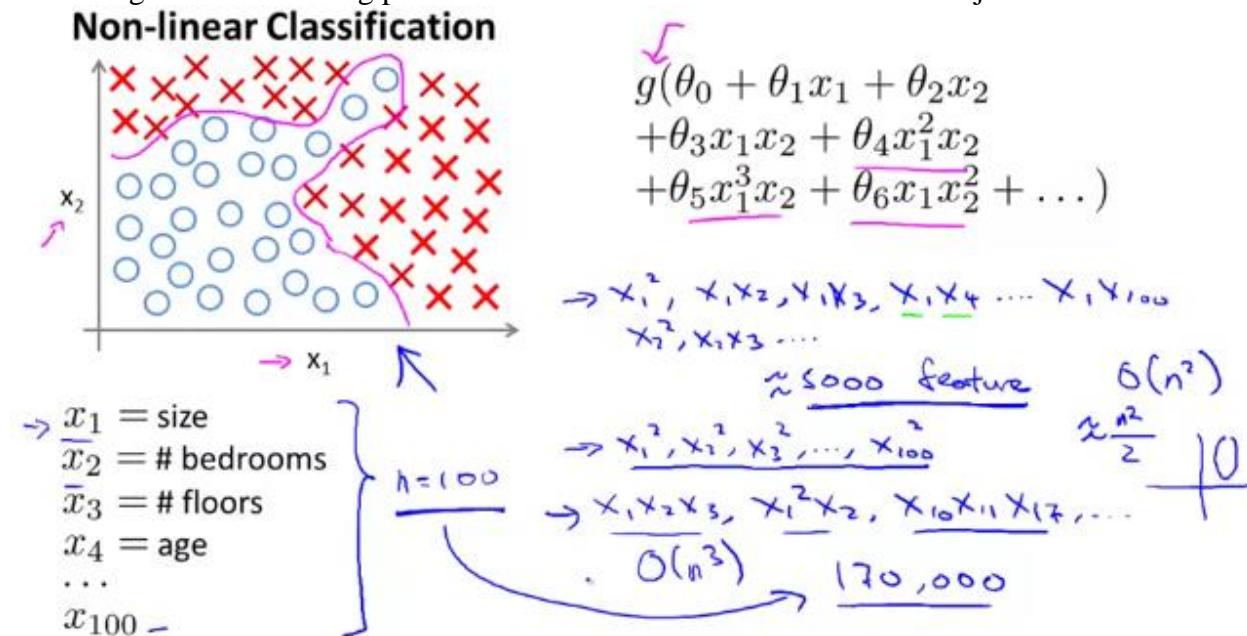
## Motivations

### Non-linear Hypotheses

In this and in the next set of videos, I'd like to tell you about a **learning algorithm** called a **Neural Network**. We're going to first talk about the representation and then in the next set of videos talk about learning algorithms for it.

Neutral networks is actually a pretty old idea, but had fallen out of favor for a while. But today, it is the state of the art technique for many different machine learning problems. So why do we need yet another learning algorithm? We already have linear regression and we have logistic regression, so **why do we need**, you know, neural networks? In order **to motivate the discussion** of neural networks, let me start by showing you a **few examples** of machine learning problems where we need to learn **complex non-linear hypotheses**.

Consider a **supervised learning classification** problem where you have a training set like this. If you want to apply logistic regression to this problem, **one thing you could do is apply logistic regression with a lot of nonlinear features** like that. So here,  $g$  as usual is the sigmoid function, and we can include lots of polynomial terms like these. And, if you include enough polynomial terms then, you know, maybe you can get a hypotheses that separates the positive and negative examples. **This particular method works well when you have only, say, two features -  $x_1$  and  $x_2$**  - because you can then include all those polynomial terms of  $x_1$  and  $x_2$ . But for many interesting machine learning problems would have a lot more features than just two.



We've been talking for a while about housing prediction, and suppose you have a housing **classification problem** rather than a regression problem, like maybe if you have different features of a house, and you want to **predict what are the odds that your house will be sold**

**within the next six months**, so that will be a classification problem. And as we saw we can come up with quite a lot of features, maybe a hundred different features of different houses. For a problem like this, **if you were to include all the quadratic terms**, all of these, even all of the quadratic **that is the second order polynomial terms**, there would be a lot of them. There would be terms like  $x_1$  squared,  $x_1x_2$ ,  $x_1x_3$ , you know,  $x_1x_4$  up to  $x_1x_{100}$  and then you have  $x_2$  squared,  $x_2x_3$  and so on. And if you include just the second order terms, that is, the terms that are a product of, you know, two of these terms,  $x_1$  times  $x_1$  and so on, then, for the case of  $n$  equals 100, **you end up with about five thousand features**. And, asymptotically, the number of quadratic features grows roughly as order  $n$  squared, where  $n$  is the number of the original features, like  $x_1$  through  $x_{100}$  that we had, and its actually closer to  $n$  squared over two. So including all the quadratic features **doesn't seem like it's maybe a good idea**, because that is a lot of features and you might end up overfitting the training set, and it can also be computationally expensive, to have, to be working with that many features. **One thing you could do is include only a subset of these**, so if you include only the features  $x_1$  squared,  $x_2$  squared,  $x_3$  squared, up to maybe  $x_{100}$  squared, then the number of features is much smaller. Here **you have only 100 such quadratic features**, but this is not enough features and certainly won't let you fit the data set like that on the upper left. In fact, if you include only these quadratic features together with the original  $x_1$ , and so on, up to  $x_{100}$  features, then you cannot actually fit very interesting hypotheses. So, you can fit things like, you know, access a line of the ellipses like these, but you certainly cannot fit a more complex data set like that shown here. So 5000 features seems like a lot, if you were to include the cubic, or third order polynomial features, the  $x_1$ ,  $x_2$ ,  $x_3$ , you know,  $x_1$  squared,  $x_2$ ,  $x_{10}$  and  $x_{11}$ ,  $x_{17}$  and so on. You can imagine there are gonna be a lot of these features. In fact, they are going to be order  $n$  cube such features and if  $n$  equals 100 you cannot compute that, you end up with on the order of about 170,000 such cubic features and so including these higher order polynomial features when your original feature set  $n$  is large this really dramatically blows up your feature space and this doesn't seem like a good way to come up with additional features with which to build non linear classifiers when  $n$  is large.

For many machine learning problems,  $n$  will be pretty large. Here's an example. Let's consider the problem of computer vision. And suppose you want to use machine learning to **train a classifier to examine an image** and tell us whether or not the image is a car.

### What is this?

You see this:



But the camera sees this:

194	210	201	212	199	213	215	195	178	158	182	209
180	189	190	221	209	205	191	167	147	115	129	163
114	126	140	188	176	165	152	140	170	106	78	88
87	103	115	154	143	142	149	153	173	101	57	57
102	112	106	131	122	138	152	147	128	84	58	66
94	95	79	104	105	124	129	119	107	87	69	67
68	71	69	98	89	92	98	95	89	88	76	67
41	56	68	99	63	45	60	82	58	76	75	65
20	43	69	75	56	41	51	73	55	70	63	44
50	50	57	69	75	75	73	74	53	68	59	37
72	59	53	66	84	92	84	74	57	72	63	42
67	61	58	65	75	78	76	73	59	75	69	50



Many people wonder why computer vision could be difficult. I mean when you and I look at this picture it is so obvious what this is. You wonder how is it that a learning algorithm could possibly fail to know what this picture is. To understand why computer vision is hard let's zoom into a small part of the image like that area where the little red rectangle is. It turns out that where you and I see a car, the computer sees that. What it sees is this matrix, or this grid of pixel intensity values that tells us the brightness of each pixel in the image. So the computer vision problem is to look at this matrix of pixel intensity values, and tell us that these numbers represent the door handle of a car.

Concretely, when we use machine learning to build a car detector, **what we do is we come up with a label training set**, with, let's say, a few label examples of cars and a few label examples of things that are not cars, then we give our training set to the learning algorithm, train a classifier and then you know, we may test it and show the new image and ask, "What is this new thing?". And hopefully it will recognize that that is a car.

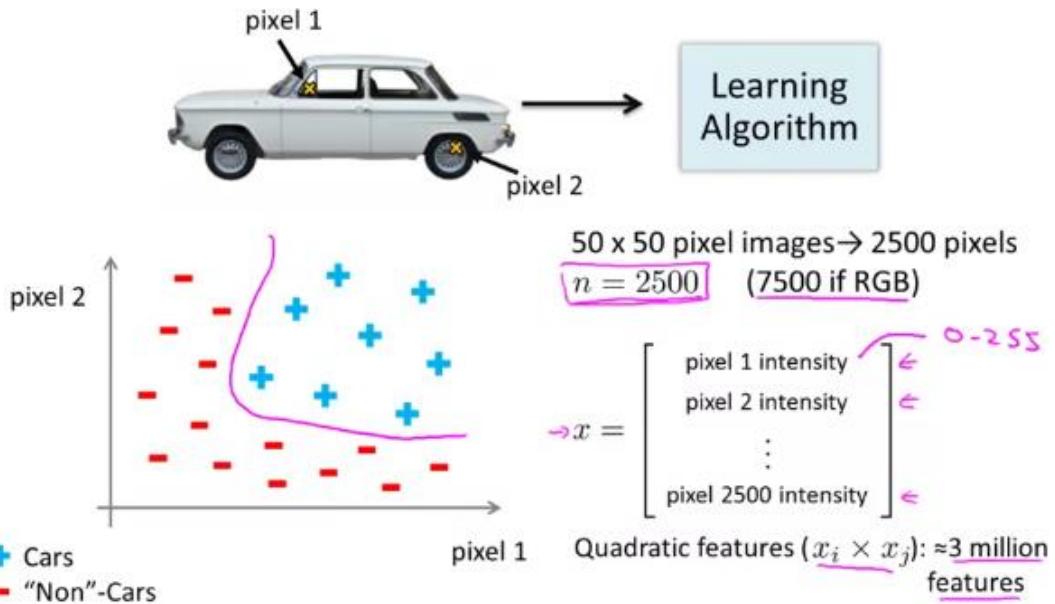
### Computer Vision: Car detection



Testing:  ←

What is this?

**To understand why we need nonlinear hypotheses**, let's take a look at some of the images of cars and maybe non-cars that we might feed to our learning algorithm.



Let's pick a couple of pixel locations in our images, so that's pixel one location and pixel two location, and let's plot this car, you know, at the location, at a certain point, depending on the intensities of pixel one and pixel two. And let's do this with a few other images. So let's take a different example of the car and you know, look at the same two pixel locations and that image has a different intensity for pixel one and a different intensity for pixel two. So, it ends up at a different location on the figure. And then let's plot some negative examples as well. That's a non-car, that's a non-car. And if we do this for more and more examples using the pluses to denote cars and minuses to denote non-cars, what we'll find is that the cars and non-cars end up lying in different regions of the space, and what we need therefore is some sort of non-linear hypotheses to try to separate out the two classes. What is the dimension of the feature space? Suppose we were to use just 50 by 50 pixel images. Now suppose our images were pretty small ones, just 50 pixels on the side. Then we would have 2500 pixels, and so the dimension of our feature size will be  $N = 2500$  where our feature vector  $x$  is a list of all the pixel intensities, you know, the pixel brightness of pixel one, the brightness of pixel two, and so on down to the pixel brightness of the last pixel where, you know, in a typical computer representation, each of these may be values between say 0 to 255 if it gives us the grayscale value. So we have  $n = 2500$ , and that's if we were using grayscale images. If we were using RGB images with separate red, green and blue values, we would have  $n = 7500$ . So, if we were to try to learn a nonlinear hypothesis by including all the quadratic features, that is all the terms of the form, you know,  $x_i \times x_j$ , while with the 2500 pixels we would end up with a total of three million features. And that's just too large to be reasonable; the computation would be very expensive to find and to represent all of these three million features per training example.

So, simple logistic regression together with adding in maybe the quadratic or the cubic features - that's just not a good way to learn complex nonlinear hypotheses when  $n$  is large because you just end up with too many features.

In the next few videos, I would like to tell you about Neural Networks, which turns out to be a much better way to learn complex hypotheses, complex nonlinear hypotheses even when your input feature space, even when  $n$  is large. And along the way I'll also get to show you a

couple of fun videos of historically important applications of Neural networks as well that I hope those videos that we'll see later will be fun for you to watch as well.

## Neurons and the Brain

Neural Networks are a pretty old algorithm that was originally motivated by the goal of having **machines that can mimic the brain**. Now in this class, of course I'm teaching Neural Networks to you because they work really well for different machine learning problems and not, certainly not because they're logically motivated.

In this video, I'd like to give you some of the background on Neural Networks. So that we can get a **sense of what we can expect them to do**. Both in the sense of applying them to modern day machinery problems, as well as for those of you that might be interested in maybe the big AI dream of someday building truly intelligent machines. Also, how Neural Networks might pertain to that.

The origins of Neural Networks was as algorithms that try to mimic the brain and those a sense that if we want to build learning systems while why not mimic perhaps the most amazing learning machine we know about, which is perhaps the brain. Neural Networks came to be very widely used throughout the 1980's and 1990's and for various reasons as popularity diminished in the late 90's.

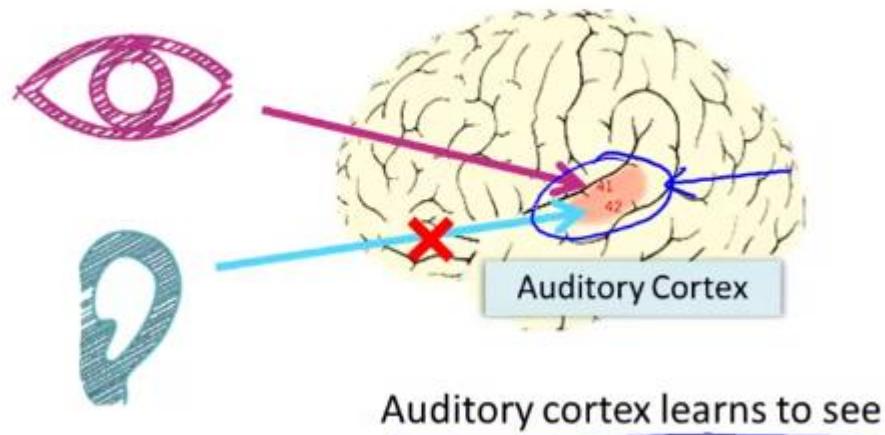
## Neural Networks

- Origins: Algorithms that try to mimic the brain.
- Was very widely used in 80s and early 90s; popularity diminished in late 90s.
- Recent resurgence: State-of-the-art technique for many applications

But more recently, Neural Networks have had a major recent resurgence. One of the reasons for this resurgence is that Neural Networks are **computationally some what more expensive algorithm** and so, it was only, you know, maybe somewhat **more recently that computers became fast enough to really run large scale Neural Networks** and because of that as well as a few other technical reasons which we'll talk about later, modern Neural Networks today are the state of the art technique for many applications.

So, when you think about mimicking the brain while one of the human brains, that itself is an amazing thing, right? The brain can learn to see, process images, learn to hear, learn to process our sense of touch. We can, you know, learn to do math, learn to do calculus, and the brain does so many different and amazing things. **It seems like if you want to mimic the brain it seems like you have to write lots of different pieces of software to mimic all of these different fascinating, amazing things that the brain does**, but there is this fascinating hypothesis that the way the brain does all of these different things is not worth like a thousand different programs, but instead, **the way the brain does is with just a single learning algorithm**.

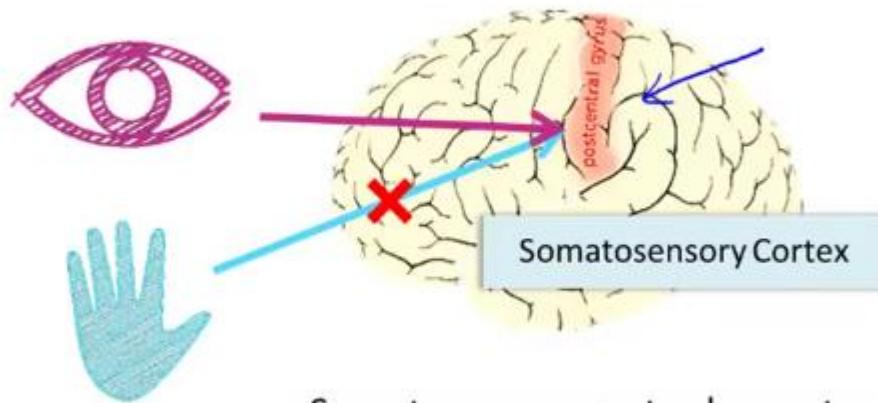
## The “one learning algorithm” hypothesis



This is just a hypothesis but let me share with you some of the evidence for this. This part of the brain that little red part of the brain, is your auditory cortex and the way you're understanding my voice now is your ear is taking the sound signal and routing the sound signal to your auditory cortex and that's what's allowing you to understand my words. Neuroscientists have done the following fascinating experiment where you **cut the wire from the ears to the auditory cortex and you re-wire**, in this case an animal's brain, **so that the signal from the eyes to the optic nerve eventually gets routed to the auditory cortex**. If you do this it turns out, **the auditory cortex will learn to see**. And this is in every single sense of the word “see” as we know it. So, if you do this to the animals, the animals can perform visual discrimination task and as they can look at images and make appropriate decisions based on the images and they're doing it with that piece of brain tissue.

Here's another example. That red piece of brain tissue is your somatosensory cortex. That's how you process your sense of touch. If you do a similar re-wiring process then the somatosensory cortex will learn to see. Because of this and other similar experiments, these are called neuro-rewiring experiments.

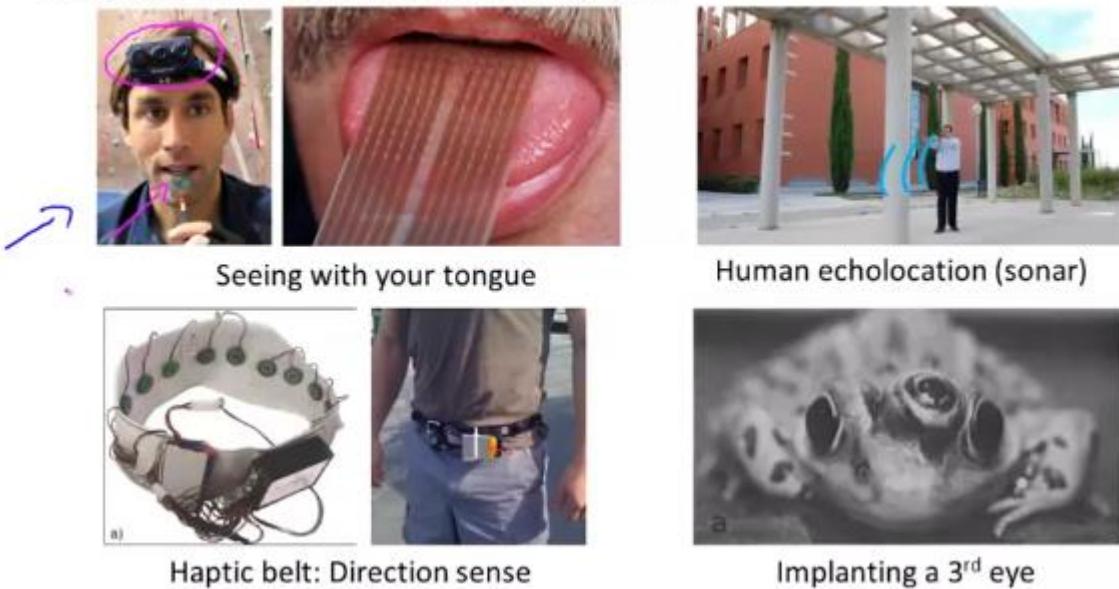
## The “one learning algorithm” hypothesis



There's this sense that if the same piece of physical brain tissue can process sight or sound or touch then maybe there is one learning algorithm that can process sight or sound or touch. And instead of needing to implement a thousand different programs or a thousand different algorithms to do, you know, the thousand wonderful things that the brain does, maybe **what we need to do is figure out some approximation or to whatever the brain's learning algorithm is** and implement that and that the brain learned by itself how to process these different types of data. To a **surprisingly large extent, it seems as if we can plug in almost any sensor to almost any part of the brain** and so, within the reason, the brain will learn to deal with it.

Here are a few more examples. On the upper left is an example of learning to see with your tongue. The way it works is--this is actually a system called **BrainPort** undergoing **FDA trials now to help blind people see**--but the way it works is, you strap a grayscale camera to your forehead, facing forward, that takes the low resolution grayscale image of what's in front of you and you then run a wire to an array of electrodes that you place on your tongue so that each pixel gets mapped to a location on your tongue where maybe a high voltage corresponds to a dark pixel and a low voltage corresponds to a bright pixel and, even as it does today, with this sort of system you and I will be able to learn to see, you know, in tens of minutes with our tongues.

### Sensor representations in the brain



Here's a second example of **human echo location** or **human sonar**. So there are two ways you can do this. You can either snap your fingers, or click your tongue. I can't do that very well. But there are blind people today that are actually being trained in schools to do this and learn to interpret the pattern of sounds bouncing off your environment - that's sonar. So, if after you search on YouTube, there are actually videos of this amazing kid who tragically because of cancer had his eyeballs removed, so this is a kid with no eyeballs. But by snapping his fingers, he can walk around and never hit anything. He can ride a skateboard. He can shoot a basketball into a hoop and this is a kid with no eyeballs.

Third example is the **Haptic Belt** where if you have a strap around your waist, ring up buzzers and **always have the north most one buzzing**. You can give a human a direction sense similar to maybe how birds can, you know, sense where north is.

And, some of the bizarre example, but if you plug a **third eye into a frog**, the frog will learn to use that eye as well.

So, it's pretty **amazing** to what extent is as if you can **plug in almost any sensor to the brain and the brain's learning algorithm will just figure out how to learn from that data and deal with that data.**

And there's a sense that if we can **figure out what the brain's learning algorithm is**, and, you know, implement it or **implement some approximation to that algorithm on a computer**, maybe that would be our best shot at, you know, making real progress towards the AI, the artificial intelligence dream of someday building truly intelligent machines.

Now, of course, I'm not teaching Neural Networks, you know, just because they might give us a window into this far-off AI dream, even though I'm personally, that's one of the things that I personally work on in my research life. But the main reason I'm teaching Neural Networks in this class is because it's actually a very effective state of the art technique for modern day machine learning applications.

So, in the next few videos, we'll start diving into the technical details of Neural Networks so that you can apply them to modern-day machine learning applications and get them to work well on problems. But for me, you know, one of the reasons the excite me is that maybe they give us this window into what we might do if we're also thinking of what algorithms might someday be able to learn in a manner similar to humankind.

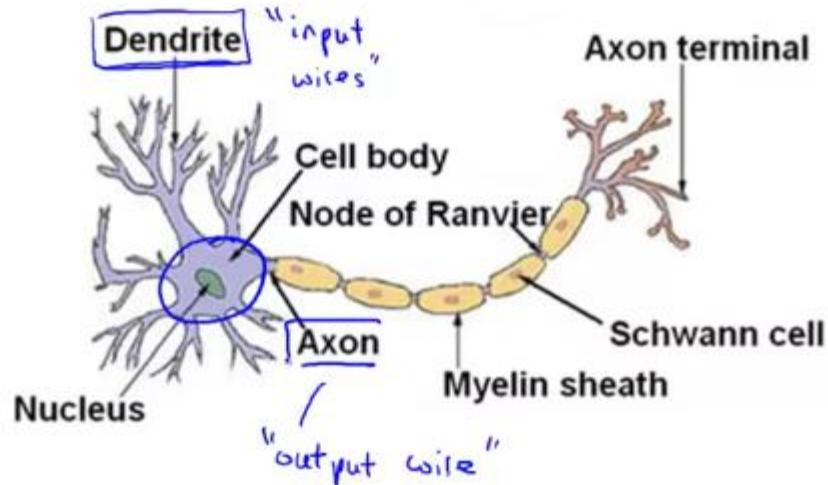
## Neural Networks

### Model Representation I

In this video, I want to start telling you about **how we represent neural networks**. In other words, **how we represent our hypothesis** or **how we represent our model** when using neural networks.

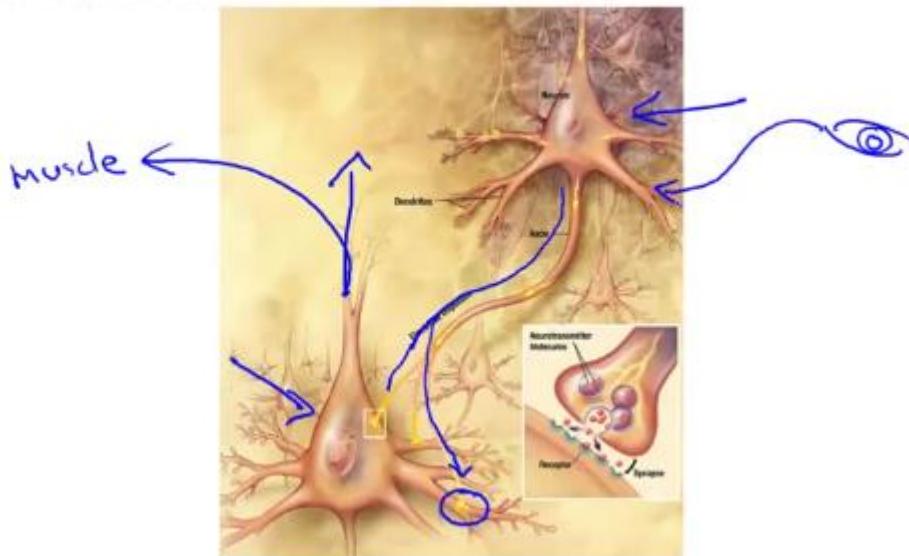
Neural networks were developed as **simulating neurons or networks of neurons in the brain**. So, to explain the hypothesis representation let's start by looking at what a single neuron in the brain looks like. **Your brain and mine is jam packed full of neurons** like these and **neurons are cells in the brain**. And two things to draw attention to are that first the **neuron has a cell body**, like so, and moreover, the **neuron has a number of input wires**, and these are called the **dendrites**, you think of them as input wires, and these receive inputs from other locations. And a **neuron also has an output wire** called an **Axon**, and this output wire is what it uses to send signals to other neurons, so to send messages to other neurons. So, at a simplistic level what a **neuron is, is a computational unit that gets a number of inputs through its input wires, does some computation, and then it sends outputs via its axon to other nodes or to other neurons in the brain.**

## Neuron in the brain



Here's an illustration of a group of neurons. **The way that neurons communicate** with each other is with **little pulses of electricity**, they are also called **spikes** but that just means pulses of electricity.

## Neurons in the brain

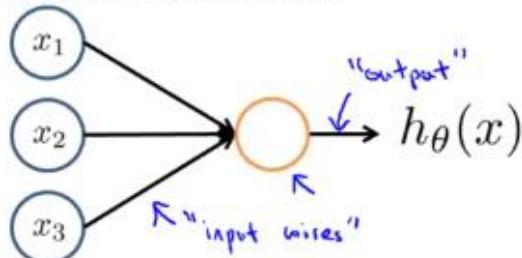


So here is **one neuron** and what it does is if it wants to send a message what it does is **sends a little pulse of electricity via its axon to some different neuron** and here, this axon that is this output wire connects to the input wire, **connects to dendrites of this second neuron over here, which then accepts this incoming message, does some computation**. And may in turn decide to send out this old message on its axon to other neurons, and this is the process by which all human thought happens. It's these **Neurons doing computations and passing messages to other neurons as a result of what other inputs they've got**. And, by the way, this is how our senses and our muscles work as well. If you want to move one of your muscles the way that works is that your neuron may send this pulses of electricity to your muscle and that causes your

muscles to contract and your eyes, some sensor like your eye must send a message to your brain, what it does it sends pulses of electricity to a neuron in your brain like so.

In a neural network, or rather, in an artificial neural network that we've implemented on the computer, we're going to use a very simple model of what a neuron does, we're going to model a neuron as just a logistic unit. So, when I draw a yellow circle like that, you should think of that as playing a role analogous, who's maybe the body of a neuron, and we then feed the neuron a few inputs via its dendrites or input wires. And the neuron does some computation. And outputs some value on this output wire, or in the biological neuron, this is an axon.

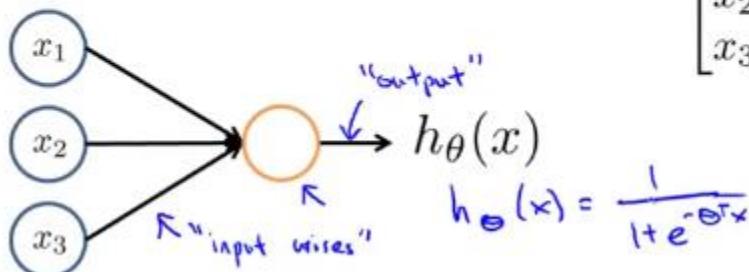
### Neuron model: Logistic unit



And whenever I draw a diagram like this, what this means is that this represents a computation of  $h(\mathbf{x})$  equals one over one plus e to the negative theta transpose  $\mathbf{x}$ , where as usual,  $\mathbf{x}$  and  $\theta$  are our parameter vectors, like so.

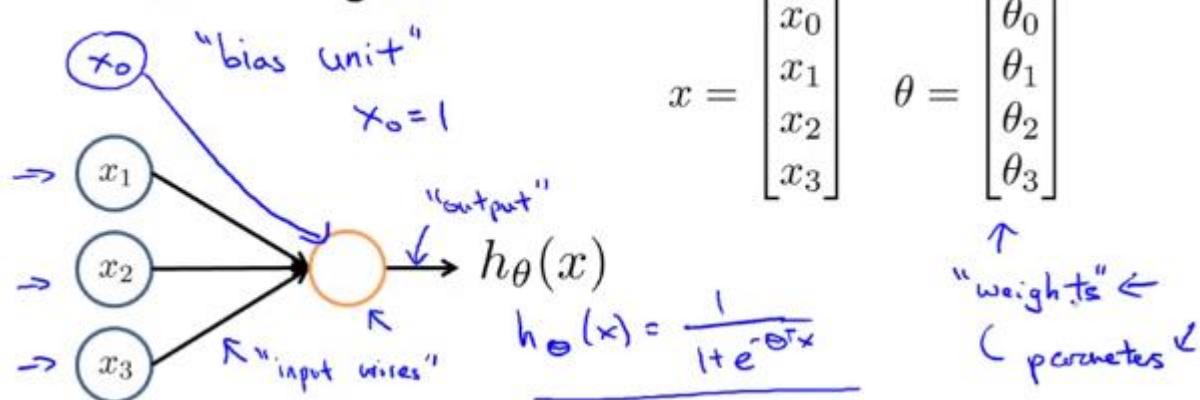
### Neuron model: Logistic unit

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$



So this is a very simple, maybe a vastly oversimplified model, of the computations that the neuron does, where it gets a number of inputs,  $x_1, x_2, x_3$  and it outputs some value computed like so. When I draw a neural network, usually I draw only the input nodes  $x_1, x_2, x_3$ .

Sometimes when it's useful to do so, I'll draw an extra node for  $x_0$ . This  $x_0$  now that's sometimes called the bias unit or the bias neuron but because  $x_0$  is already equal to 1, sometimes, I draw this, sometimes I won't just depending on whatever is more notationally convenient for that example.

**Neuron model: Logistic unit**

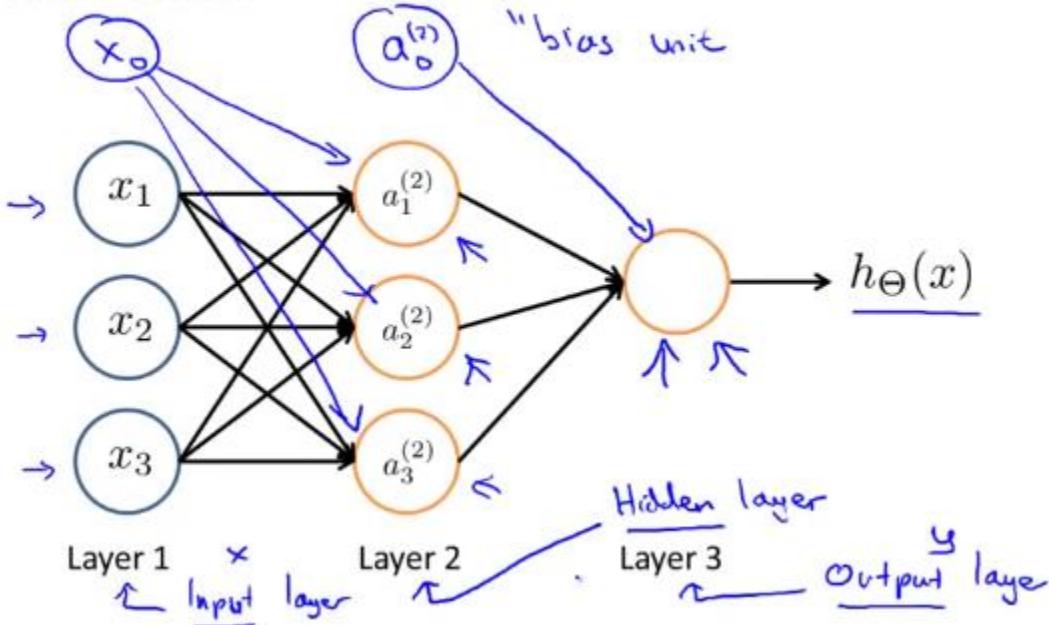
Finally, one last bit of terminology when we talk about neural networks, sometimes we'll say that this is a **neuron** or an **artificial neuron** with a **Sigmoid** or **logistic activation function**. So this **activation function** in the neural network terminology, this is **just another term for that function** for that non-linearity  $g(z) = 1$  over  $1+e$  to the  $-z$ .

**Sigmoid (logistic) activation function.**

$$g(z) = \frac{1}{1+e^{-z}}$$

And whereas so far I've been calling  $\theta$  the parameters of the model, I'll mostly continue to use that terminology to call  **$\theta$ , the parameters**, but in neural networks, in the neural network literature sometimes you might hear people talk about **weights of a model** and the weights just means exactly the same thing as parameters of a model. But I'll mostly continue to use the terminology **parameters** in these videos, but sometimes, you might hear others use the **weights** terminology.

So, this little **diagram represents a single neuron**.

**Neural Network**

What a **neural network is, is just a group of this different neurons strung together.**

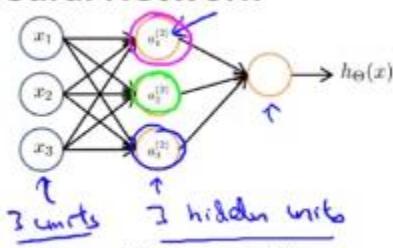
Concretely, here we have **input units  $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$**  and once again, sometimes you can draw this extra note  $x_0$  and sometimes not, just flow that in here. And here we have **three neurons**, which I have written  $\mathbf{a}^{(2)}_1, \mathbf{a}^{(2)}_2, \mathbf{a}^{(2)}_3$ . I'll talk about those indices later. And once again you know we can if we want add in just  $\mathbf{a}^{(2)}_0$  as an **extra bias unit** there. There's always, outputs a value of 1. And then finally we have this **third node at the final layer**, and there's this **third node that outputs the value** that the **hypothesis  $h(\mathbf{x})$  computes**. To introduce a bit more terminology, in a neural network, the first layer, this is also called the **input layer** because this is where we Input our features,  $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ . The **final layer** is also called the **output layer** because that layer has the neuron, this one over here that outputs the final value computed by a hypothesis. And then, layer 2 in between, this is called the **hidden layer**. The term hidden layer isn't a great terminology, but the intuition is that, you know, in supervised learning, where you get to see the inputs and you get to see the correct outputs, where the hidden layer of values you don't get to observe in the training sets. It's not  $\mathbf{x}$ , and it's not  $y$ , and so we call those **hidden**. And there are times when we see neural networks with more than one hidden layer but in this example, we have one input layer, Layer 1, one hidden layer, Layer 2, and one output layer, Layer 3. But basically, anything that isn't an input layer and isn't an output layer is called a **hidden layer**.

So I want to be really clear about **what this neural network is doing**.

Let's step through the computational steps that are embodied by this, represented by this diagram. To explain these specific computations represented by a neural network, here's a little bit more notation. I'm going to use  $\mathbf{a}^{(j)}_i$  to denote the activation of **neuron i** or of **unit i** in **layer j**. So concretely, this  $\mathbf{a}^{(2)}_1$ , that's the activation of the **first unit in layer two**, in our hidden layer.

And **by activation I just mean the value that's computed by and as output by a specific neuron**. In addition, neural network is **parameterized by these matrixes**,  $\boldsymbol{\theta}^j$ , where  $\boldsymbol{\theta}^j$  is going to be a **matrix of weights** controlling the **function mapping** form one layer, maybe the first layer to the second layer, or from the second layer to the third layer.

## Neural Network



$\rightarrow a_i^{(j)}$  = "activation" of unit  $i$  in layer  $j$

$\rightarrow \Theta^{(j)}$  = matrix of weights controlling function mapping from layer  $j$  to layer  $j + 1$

$$\Theta^{(j)} \in \mathbb{R}^{3 \times 4}$$

$$\rightarrow a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3)$$

$$\rightarrow a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$$

$$\rightarrow a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)$$

$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

$\Rightarrow$  If network has  $s_j$  units in layer  $j$ ,  $s_{j+1}$  units in layer  $j + 1$ , then  $\Theta^{(j)}$  will be of dimension  $s_{j+1} \times (s_j + 1)$ .

So here are the computations that are represented by this diagram.

This first hidden unit, here is its value computed as follows, there's  $a^{(2)}_1$  equal to the **sigmoid function** or **the sigma activation function**, also called the **logistics activation function**, apply to this sort of a linear combination of these inputs. And then this second hidden unit has this activation value computed as sigmoid of this, and similarly for this third hidden unit is computed by that formula. So here we have 3 input units and the 3 hidden units, and so the dimension of  $\Theta^{(1)}$ , which is the matrix of parameters governing our mapping from our three input units to our hidden units.  $\Theta^{(1)}$  is going to be a 3,  $\Theta^{(1)}$  is going to be a **3x4-dimensional matrix**. And more generally, if a network has  **$s_j$  units in layer  $j$**  and  **$s_j + 1$  units in layer  $j + 1$**  then the matrix  $\Theta^{(j)}$  which governs the function mapping from layer  $j$  to layer  $j + 1$ , that will have to mention  **$s_{j+1}$**  by  **$s_j + 1$** , just be clear about this notation right. This is  **$s$  subscript  $j + 1$**  and that's  **$s$  subscript  $j$** , **and then this whole thing, plus 1**, this whole thing ( **$s_j + 1$** ), okay? So that's  **$s$  subscript  $j + 1$**  plus, by, So that's  **$s$  subscript  $j + 1$  by  $s_j + 1$**  where this plus one is not part of the subscript.

Okay, so we talked about **what the three hidden units do to compute their values**. Finally, there's a last, this final in output layer we have one more unit which computes  $h(x)$  and that's equal, can also be written as  $a^{(3)}_1$ , and that's equal to this. And you notice that I've written this with a superscript two here, because **theta of superscript two is the matrix of parameters**, or **the matrix of weights** that **controls the function that maps from the hidden units**, that is the layer two units to the one layer three unit, that is the output unit. To summarize, what we've done is shown how a picture like this over here defines an artificial neural network which defines a **function  $h$  that maps from  $x$ 's, input values**, to hopefully, to some space of **predictions  $y$** . And these hypothesis are **parameterized by parameters denoted with a capital theta** so that, as we vary theta, we get different hypothesis and we get different functions mapping say from  $x$  to  $y$ . So this gives us **a mathematical definition of how to represent the hypothesis in the neural network**.

In the next few videos what I would like to do is give you more intuition about what these hypothesis representations do, as well as go through a few examples and talk about how to compute them efficiently.

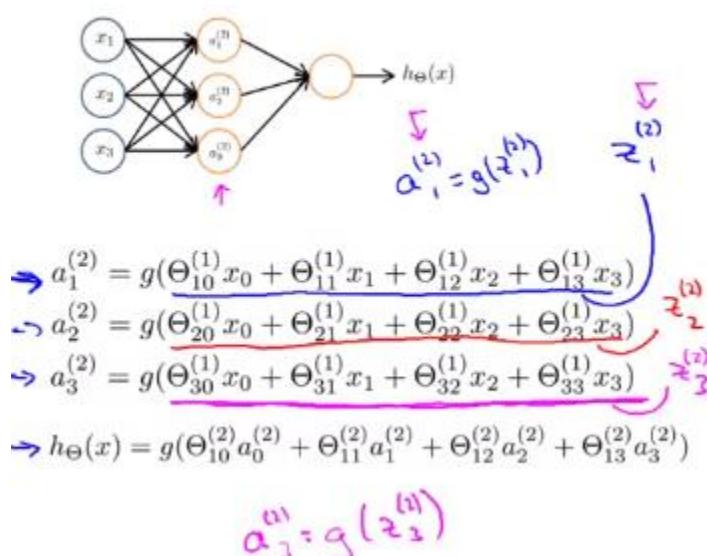
## Model Representation II

In the last video, we gave a **mathematical definition of how to represent** or how to compute the hypotheses used by Neural Network.

In this video, I like show you **how to actually carry out that computation** efficiently, and that is show you **a vectorized implementation**. And second, and more importantly, I want to start giving you intuition about **why these neural network representations might be a good idea** and **how they can help us to learn complex nonlinear hypotheses**.

Consider this neural network. Previously we said that the sequence of steps that we need in order to compute the output of a hypotheses is **these equations** given on the left where we **compute** the **activation values** of the **three hidden** units and then we use those to compute the **final output** of our hypotheses  $h_{\Theta}(x)$ .

### Forward propagation: Vectorized implementation



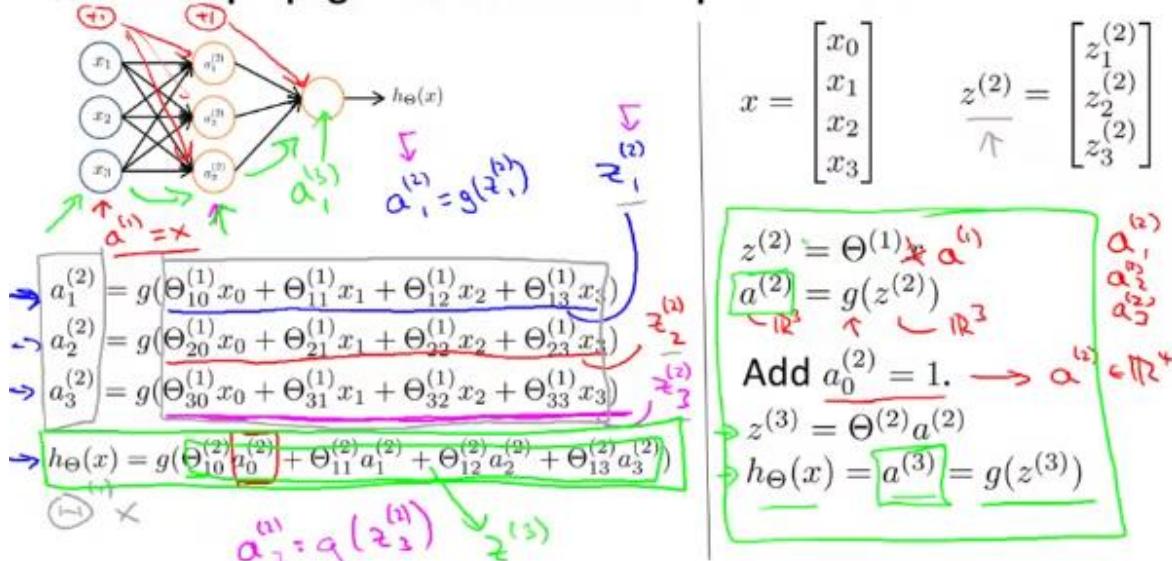
Now, I'm going to **define a few extra terms**. So, this term that I'm underlining here, I'm going to define that to be  $\mathbf{z}^{(2)}_1$ , so that we have that  $\mathbf{a}^{(2)}_1$ , which is this term, is equal to  $g(\mathbf{z}^{(2)}_1)$ . And by the way, these superscript 2, you know, what that means is that the  $\mathbf{z}^{(2)}$  and this  $\mathbf{a}^{(2)}$  as well, the superscript 2 in parentheses means that these are **values associated with layer 2**, that is with the **hidden layer** in the neural network. Now this term here, I'm going to similarly define as  $\mathbf{z}^{(2)}_2$ . And finally, this last term here that I'm underlining, let me define that as  $\mathbf{z}^{(2)}_3$ . So that similarly we have  $\mathbf{a}^{(2)}_3$  equals  $g(\mathbf{z}^{(2)}_3)$ . So these **z values** are just a linear combination, **a weighted linear combination**, of the input values  $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$  that go into a particular neuron.

Now if you look at this block of numbers, you may notice that that block of numbers corresponds suspiciously similar to the matrix vector operation, **matrix vector multiplication of  $\theta_1$  times the vector  $x$** .

$$\begin{aligned} & g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3) \\ & g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3) \\ & g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3) \end{aligned}$$

Using this observation **we're going to be able to vectorize this computation** of the neural network.

### Forward propagation: Vectorized implementation



Concretely, let's define the **feature vector  $x$**  as usual to be the vector of  $x_0, x_1, x_2, x_3$  where  $x_0$  as usual is always equal to 1 and let's define  $z^{(2)}$  to be the **vector of these  $z$ -values**, you know, of  $z_1^{(2)}, z_2^{(2)}, z_3^{(2)}$ . And notice that, there,  $z^{(2)}$  this is a **three dimensional vector**. We can now vectorize the computation of  $a_1^{(2)}, a_2^{(2)}, a_3^{(2)}$  as follows.

We can just write this in two steps. We can compute  $z^{(2)}$  as  $\theta_1$  times  $x$  and that would give us this vector  $z^{(2)}$ ; and then  $a^{(2)}$  is  $g(z^{(2)})$ , and just to be clear  $z^{(2)}$  here, this is a **three-dimensional vector** and  $a^{(2)}$  is also a **three-dimensional** vector and thus this **activation  $g$**  this **applies the sigmoid function element-wise to each of the  $z^{(2)}$ 's elements**. And by the way, to make our notation a little more consistent with what we'll do later, in this input layer we have the inputs  $x$ , but we can also think of this as the activations of the first layers. So, if I defined  $a^{(1)}$  to be equal to  $x$ , so that  $a^{(1)}$  is a vector, I can now take this  $x$  here and replace this with  $z^{(2)}$  equals  $\theta_1$  times  $a^{(1)}$  just by defining  $a^{(1)}$  to be activations in my input layer.

### Vectorizing the computation

- **Parameters or Weights** -  $\theta^j$  – **Matrix of weights** controlling function mapping from layer  $j$  to layer  $j+1$ 
  - $\theta^{(1)}$  - parameters controlling function mapping between **layer 1 and 2**
  - $\theta^{(2)}$  - parameters controlling function mapping between **layer 2 and 3**

- **Input Layer 1 - Three input units –  $x_1, x_2, x_3$**  and once again, sometimes extra  $x_0$  (bias unit). Let's define  $x$  as **Input Feature Vector** consisting of input values  $x_0, x_1, x_2, x_3$  that go into a particular neuron;  $x_0$  as usual is always equal to 1

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

- **Output of layer 2:** Hidden layer, there are three neurons namely –

$a^{(2)}_1, a^{(2)}_2, a^{(2)}_3$  Layer 2 unit 1, 2, 3.

We can, if we want, add in  $a^{(2)}_0$  as an **extra bias unit**.

**Output values of:**  $a^{(2)}_1, a^{(2)}_2, a^{(2)}_3$  will be inputs to layer 3, and will be used to compute the **final output** of hypotheses  $h(x)$ .

$$a^{(2)}_1 = g(\theta^{(1)}_{10} x_0 + \theta^{(1)}_{11} x_1 + \theta^{(1)}_{12} x_2 + \theta^{(1)}_{13} x_3) = g(z^{(2)}_1)$$

$$a^{(2)}_2 = g(\theta^{(1)}_{20} x_0 + \theta^{(1)}_{21} x_1 + \theta^{(1)}_{22} x_2 + \theta^{(1)}_{23} x_3) = g(z^{(2)}_2)$$

$$a^{(2)}_3 = g(\theta^{(1)}_{30} x_0 + \theta^{(1)}_{31} x_1 + \theta^{(1)}_{32} x_2 + \theta^{(1)}_{33} x_3) = g(z^{(2)}_3)$$

These **z values** are just a weighted linear combination, of the input values  $x_0, x_1, x_2, x_3$  that go into a particular neuron. You may notice that the block of numbers  $z^{(2)}_1, z^{(2)}_2, z^{(2)}_3$  correspond similar to **matrix vector multiplication**,  $\theta_1$ (matrix) times  $x$  (vector).

Let's define  $z^{(2)}$  to be a vector – consisting of these 3 z values  $z^{(2)}_1, z^{(2)}_2, z^{(2)}_3$

$$z^{(2)} = \underbrace{\begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix}}$$

We can now **vectorize the computation** of  $a^{(2)}_1, a^{(2)}_2, a^{(2)}_3$  as follows:

[Important Note: Since we are using notation  $a^{(2)}$  as activation of 2<sup>nd</sup> layer, to make our notation a little more consistent, though in the input layer we have the inputs  $x$ , but we can also think of this  $x$  as the activations of the first layer and as  $a^{(1)}$ , so that  $a^{(1)}$  is a vector and we can replace  $x$  with  $a^{(1)}$ ].

$z^{(2)} = \theta^{(1)} x = \theta^{(1)} a^{(1)}$  - This gives us vector  $z^{(2)}$  consisting of  $z^{(2)}_1, z^{(2)}_2, z^{(2)}_3$ .

$a^{(2)} = g(z^{(2)})$  - Where  $a^{(2)}$  is also a four-dimensional feature vector comprised of  $a^{(2)}_0, a^{(2)}_1, a^{(2)}_2, a^{(2)}_3$ . Note that this activation  $g$  applies the sigmoid function element-wise to each of the  $z^{(2)}$ 's elements

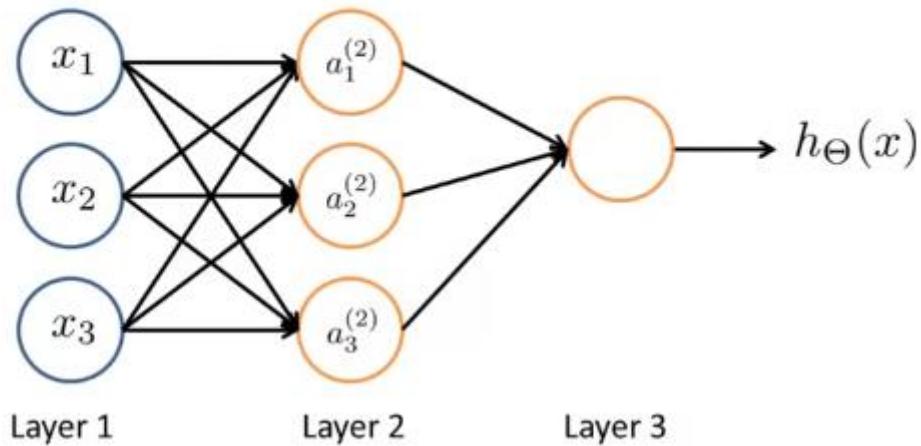
- **Output of layer 3:**  $z^{(3)} = \theta^{(2)} a^{(2)}$   
 $h_\theta(x) = a^{(3)} = g(\theta^{(2)}_{10} a^{(2)}_0 + \theta^{(2)}_{11} a^{(2)}_1 + \theta^{(2)}_{12} a^{(2)}_2 + \theta^{(2)}_{13} a^{(2)}_3) = g(z^{(3)})$

Now, with what I've written so far, I've now gotten myself the values for  $a^{(2)}_1, a^{(2)}_2, a^{(2)}_3$  and really I should put the superscripts there as well. But I need one more value, which is I also want this  $a^{(2)}_0$  and that corresponds to a **bias unit** in the **hidden layer** that goes to the output there.

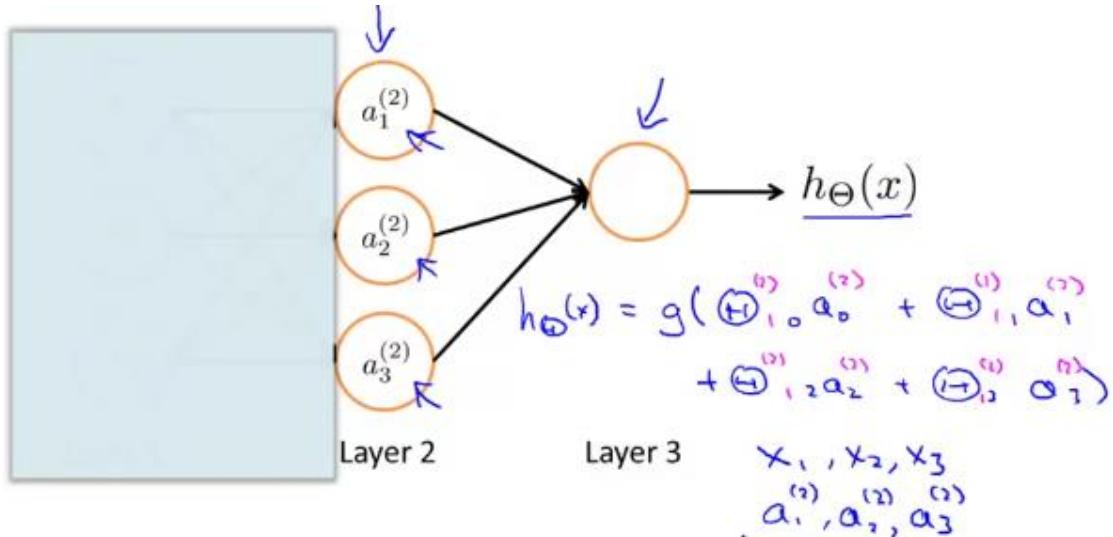
Of course, there was a bias unit here too that, you know, I just didn't draw under here, but to take care of this extra bias unit, what we're going to do is add an extra  $a_0$  superscript 2, that's equal to one, and after taking this step we now have that  $\mathbf{a}^{(2)}$  is going to be a **four dimensional feature vector** because we just added this extra, you know,  $a_0$  which is equal to 1 corresponding to the bias unit in the hidden layer.

And finally, to **compute the actual value output of our hypotheses**, we then simply need to compute  $\mathbf{z}^{(3)}$ . So  $\mathbf{z}^{(3)}$  is equal to this term here that I'm just underlining, this inner term here is  $\mathbf{z}^{(3)}$ . And  $\mathbf{z}^{(3)}$  is stated  $\mathbf{0}^{(2)}$  times  $\mathbf{a}^{(2)}$  and finally my hypotheses output  $h(x)$  which is  $\mathbf{a}^{(3)}$  that is the activation of my one and only unit in the output layer. So, **that's just the real number**. You can write it as  $\mathbf{a}^3$  or as  $\mathbf{a}^{(3)}_1$  and that's  $\mathbf{g}(\mathbf{z}^{(3)})$ . This process of computing  $h(x)$  is also called **forward propagation** and is called that because we start off with the activations of the input-units and then we sort of forward-propagate that to the hidden layer and compute the activations of the hidden layer and then we sort of forward propagate that and compute the activations of the output layer, but this process of computing the activations from the input then the hidden then the output layer, and that's also called forward propagation and what we just did is we just worked out a vector wise implementation of this procedure. So, if you implement it using these equations that we have on the right, these would give you an efficient way of computing  $h(x)$ .

### Neural Network learning its own features

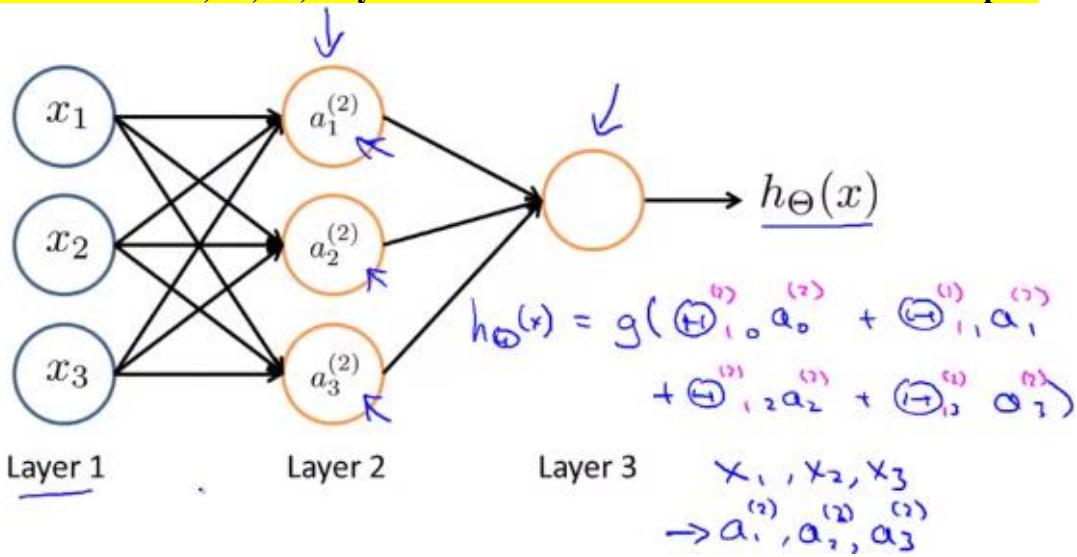


This forward propagation view also helps us to understand what Neural Networks might be doing and why they might help us to learn interesting nonlinear hypotheses. Consider the following neural network and let's say I cover up the left path of this picture for now.



If you look at what's left in this picture, this looks a lot like logistic regression where what we're doing is we're using that node, that's just the logistic regression unit and we're using that to make a prediction  $h(x)$ . And concretely, what the hypotheses is outputting is  $h(x)$  is going to be equal to  $g$  which is my **sigmoid activation** function times  $\Theta_0$  times  $a_0$  is equal to 1, plus  $\Theta_1$  times  $a_1$ , plus  $\Theta_2$  times  $a_2$ , plus  $\Theta_3$  times  $a_3$ , where values  $a_1, a_2, a_3$  are those given by these three given units.

Now, to be actually consistent to my earlier notation, actually, we need to, you know, fill in these superscript 2's here everywhere and I also have these indices 1 there because I have only one output unit, but if you focus on the blue parts of the notation this is, you know, this looks awfully like the standard logistic regression model, except that I now have a capital theta instead of lower case theta. And what this is doing is just logistic regression. But where the features fed into logistic regression are these values computed by the hidden layer. Just to say that again, **what this neural network is doing is just like logistic regression**, except that rather than using the original features  $x_1, x_2, x_3$ , is using these new features  $a_1, a_2, a_3$ . Again, we'll put the superscripts there, you know, to be consistent with the notation. And **the cool thing about this, is that the features  $a_1, a_2, a_3$ , they themselves are learned as functions of the input.**

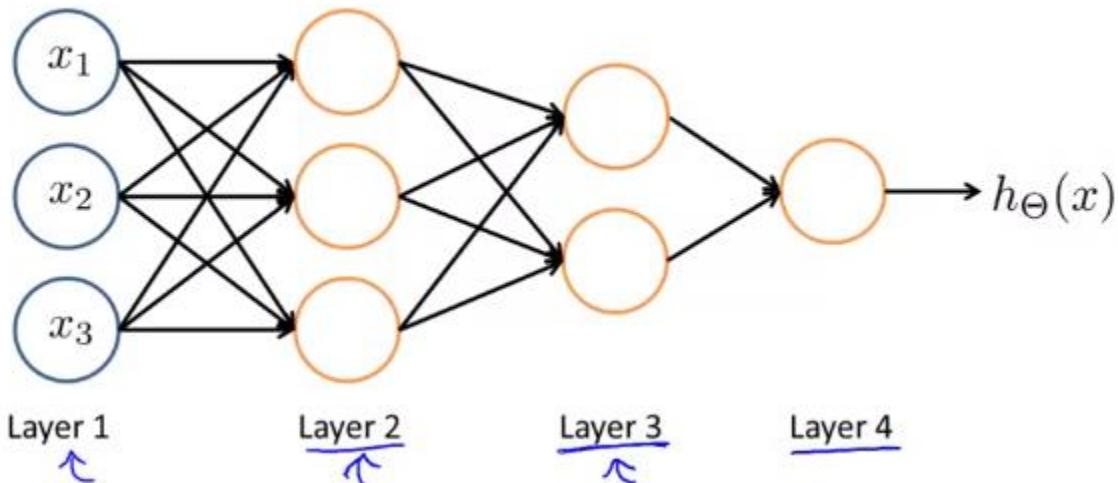


Concretely, the **function mapping from layer 1 to layer 2, that is determined by some other set of parameters, theta 1**. So it's as if the neural network, **instead of being constrained to feed the features  $x_1, x_2, x_3$  to logistic regression, it gets to learn its own features,  $a_1, a_2, a_3$** , to feed into the logistic regression and as you can imagine depending on what parameters it chooses for theta 1 you can learn some pretty interesting and complex features and therefore you can end up with a better hypotheses than if you were constrained to use the raw features  $x_1, x_2$  or  $x_3$  or if you will constrain to say choose the polynomial terms, you know,  $x_1, x_2, x_3$ , and so on. But instead, **this algorithm has the flexibility to try to learn whatever features it wants**, using these  $a_1, a_2, a_3$  in order to feed into **this last unit that's essentially a logistic regression here**.

I realized this example is described at somewhat high level and so I'm not sure if this intuition of the neural network, you know, having more complex features will quite make sense yet, but if it doesn't yet in the next two videos I'm going to go through a specific example of how a neural network can use this hidden layer to compute more complex features to feed into this final output layer and how that can learn more complex hypotheses.

So, in case what I'm saying here doesn't quite make sense, stick with me for the next two videos and hopefully after working through those examples this explanation will make a little bit more sense. But just the point O. You can have neural networks with other types of diagrams as well, and **the way that neural networks are connected, that's called the architecture**. So the term architecture refers to how the different neurons are connected to each other.

### Other network architectures



This is an example of a different neural network architecture and once again you may be able to get this intuition of how the second layer, here we have three heading units that are computing some complex function maybe of the input layer, and then the **third layer can take the second layer's features and compute even more complex features in layer three so that by the time you get to the output layer, layer four, you can have even more complex features** of what you are able to compute in layer three and so get very interesting nonlinear hypotheses.

By the way, in a network like this, layer one, this is called an input layer. Layer four is still our output layer, and this network has two hidden layers. So anything that's not an input layer or an output layer is called a hidden layer. So, hopefully from this video you've gotten a sense of how

the **feed forward propagation** step in a neural network works where you **start from the activations of the input layer and forward propagate that to the first hidden layer, then the second hidden layer, and then finally the output layer.** And you also saw how we can vectorize that computation.

In the next, I realize that some of the intuitions in this video of how, you know, other certain layers are computing complex features of the earlier layers. I realized some of that intuition may be still slightly abstract and kind of a high level.

And so what I would like to do in the two videos is **work through a detailed example of how a neural network can be used to compute nonlinear functions** of the input and hope that will give you a good sense of the sorts of complex nonlinear hypotheses we can get out of Neural Networks.

## Applications

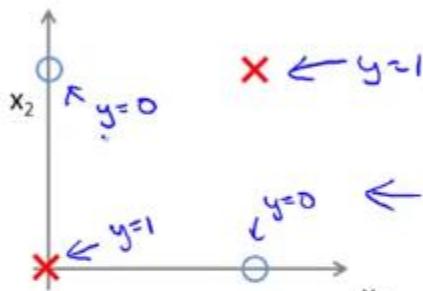
### Examples and Intuitions I

In this and the next video I want to work through a detailed example showing **how a neural network can compute a complex non linear function of the input.** And hopefully this will give you a good sense of why neural networks can be used to learn complex non linear hypotheses.

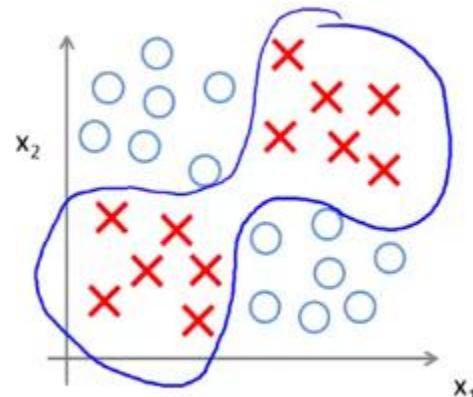
Consider the following problem where we have input features  $x_1$  and  $x_2$  that are **binary values**. So, either 0 or 1. So,  $x_1$  and  $x_2$  **can each take on only one of two possible values**. In this example, I've drawn only two positive examples and two negative examples. That you can think of this as a simplified version of a more complex learning problem where we may have a bunch of **positive examples in the upper right and lower left**, and a bunch of **negative examples denoted by the circles**. And what we'd like to do is learn a **non-linear division of boundary** that may need to separate the positive and negative examples.

#### Non-linear classification example: XOR/XNOR

→  $x_1, x_2$  are binary (0 or 1).



$$\begin{aligned} y &= \underline{x_1 \text{ XOR } x_2} \\ &\rightarrow \underline{x_1 \text{ XNOR } x_2} \\ &\hookrightarrow \underline{\text{NOT } (x_1 \text{ XOR } x_2)} \end{aligned}$$



So, how can a neural network do this, and rather than using the example on the right, we are going to use this, maybe easier to examine example on the left.

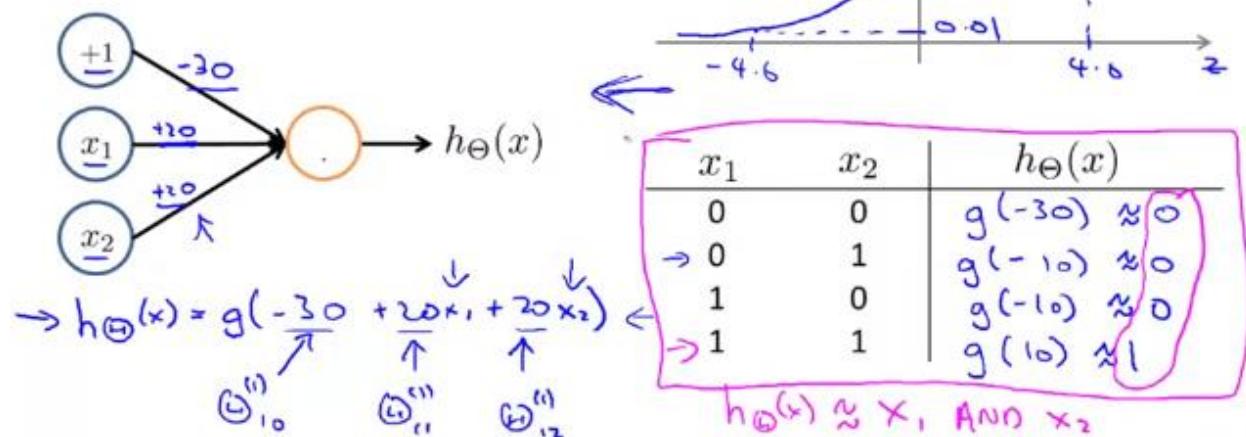
Concretely what this is, is really computing the value of label  $y$  equals  $x_1 \text{ XOR } x_2$  or actually this is actually  $x_1 \text{ XNOR } x_2$  function where  $\text{XNOR}$  is the **alternative notation** for  $\text{NOT}(x_1 \text{ XOR } x_2)$ . So,  $x_1 \text{ XOR } x_2$  that's **true only if** exactly one of  $x_1$  or  $x_2$  is equal to 1. It turns out that these specific examples works out a little bit better if we use the XNOR example instead, these two are the same of course, this means  $\text{NOT}(x_1 \text{ XOR } x_2)$  and so we're going to have positive examples of either both are true or both are false, and what we have as  $y$  equals 1,  $y$  equals 1. And we're going to have  **$y$  equals 0 if only one of them is true**, and we're going to figure out if we can get a neural network to fit to this sort of training set.

[**Explanation:**] We can see that in the left diagram, at the points where both  $x_1$  and  $x_2$  are zero i.e. at origin, there  $y = 1$ , next we see that where  $x_1$  is 1 and  $x_2$  is 0 i.e. on  $x_1$  axis, there  $y = 0$ ; and when  $x_1$  and  $x_2$  both are one, there  $y = 1$ . Thus we have  $y = 1$  only when both  $x_1$  or  $x_2$  are 1 (i.e. true) or both are 0 (i.e. false)]

In order to build up to a network that fits the XNOR example we're going to **start with a slightly simpler one and show a network that fits the AND function**.

### Simple example: AND

- $x_1, x_2 \in \{0, 1\}$
- $y = x_1 \text{ AND } x_2$



Concretely, let's say we have input  $x_1$  and  $x_2$  that are again binaries so, it's either 0 or 1 and let's say our target labels  $y = x_1 \text{ AND } x_2$ , this is a logical AND. So, can we get a one-unit network to compute this logical AND function? In order to do so, I'm going to actually draw in the bias unit as well, the plus one unit. Now let me just assign some values to the weights or parameters of this network. I'm gonna write down the parameters on this diagram here,  $-30$  here.  $+20$  and  $+20$ . And what this mean is just that I'm assigning a value of  $-30$  to the value associated with  $x_0$ , this  $+1$  going into this unit, and a parameter value of  $+20$  that multiplies to  $x_1$ , a value of  $+20$  for the parameter that multiplies into  $x_2$ .

So, concretely it's the same that the hypothesis  $h(x) = g(-30 + 20x_1 + 20x_2)$ . So, sometimes it's just convenient to draw these weights. Draw these **parameters** up here in the diagram within neural network, and of course this- 30, this is actually  $\theta^{(1)}_{10}$ , this is  $\theta^{(1)}_{11}$ , and that's  $\theta^{(1)}_{12}$  but it's just easier to think about it as associating these parameters with the edges of the network.

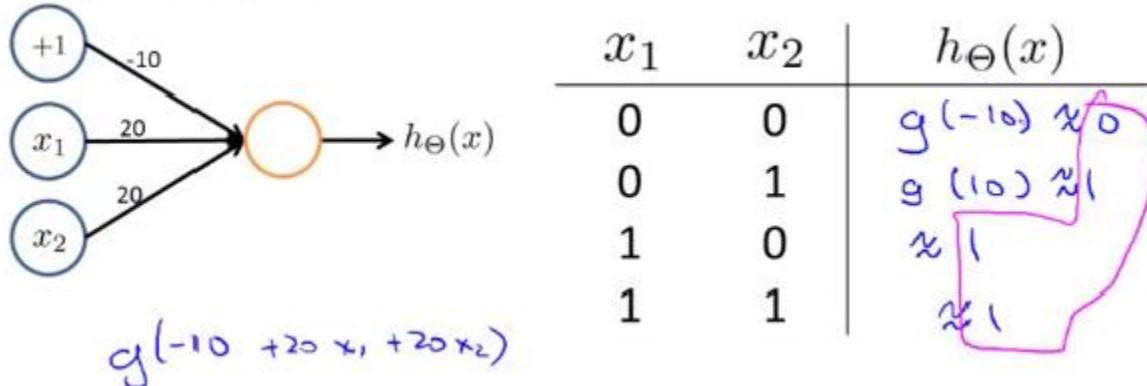
Let's look at **what this little single neuron network will compute**. Just to remind you the **sigmoid activation function  $g(z)$  looks like this**. It starts from 0 rises smoothly crosses 0.5 and then it asymptotes at 1, and to give you some landmarks, if the horizontal axis value  $z$  is equal to 4.6 then the sigmoid function is equal to 0.99. This is **very close to 1** and kind of symmetrically, if it's -4.6 then the sigmoid function there is 0.01 which is **very close to 0**. Let's look at the **four possible input values for  $x_1$  and  $x_2$**  and look at what the hypotheses will output in that case.

If  $x_1$  and  $x_2$  are both equal to 0, if you look at this, if  **$x_1$  and  $x_2$  are both equal to 0** then the **hypothesis outputs  $g(-30)$** , so, this is very far to the left of this diagram, so it will be very close to 0.

If  $x_1$  equals 0 and  $x_2$  equals 1, then this formula here evaluates the  $g$  that is the sigmoid function applied to -10, and again that's you know to the far left of this plot and so, that's again very close to 0. This is also  $g$  of minus 10 that is if  $x_1$  is equal to 1 and  $x_2$  is 0, this minus 30 plus 20 which is minus 10, and finally if  $x_1$  equals 1  $x_2$  equals 1 then you have  $g$  of minus 30 plus 20 plus 20. So, that's  $g$  of positive 10 which is therefore very close to 1. And if you look in this column this is exactly the **logical AND function**. So, this is computing  $h(x)$ , is approximately  **$x_1$  AND  $x_2$** . In other words, it outputs 1 if and only if  $x_2$ ,  $x_1$  and  $x_2$ , are both equal to 1. So, by writing out our little **truth table** like this we manage to figure out what's the logical function that our neural network computes.

This network showed here **computes the OR function**.

### Example: OR function



Just to show you how I worked that out. If you are, write out the hypothesis if you find this confusing,  $g(-10 + 20x_1 + 20x_2)$  and so you fill in these values. You find that's  $g$  of minus 10 which is approximately 0,  $g$  of 10 which is approximately 1, and so on and these are approximately 1 and approximately 1 and these numbers are essentially the **logical OR function**.

So, hopefully with this you now understand how single neurons in a neural network can be used to compute logical functions like AND and OR and so on. In the next video we'll continue

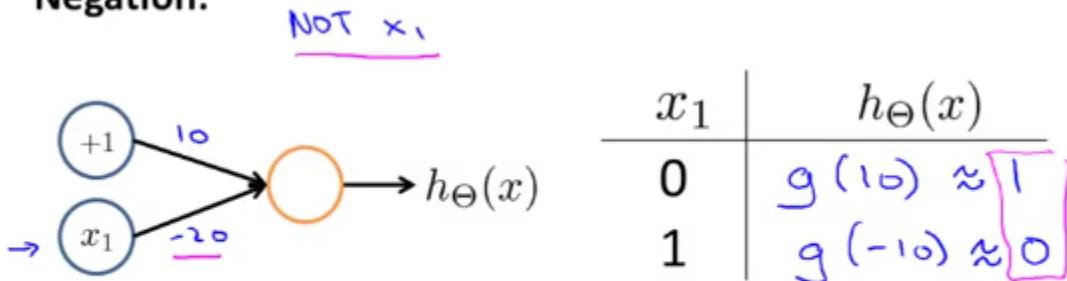
building on these examples and work through a more complex example. We'll get to show you how a neural network now with multiple layers of units can be used to compute more complex functions like the XOR function or the XNOR function.

## Examples and Intuitions II

In this video I'd like to keep working through our example to show how a Neural Network can compute complex non linear hypothesis. In the last video we saw how a Neural Network can be used to compute the functions  $x_1$  AND  $x_2$ , and the function  $x_1$  OR  $x_2$  when  $x_1$  and  $x_2$  are binary, that is when they take on values 0,1.

We can also have a network to compute negation that is to compute the function NOT  $x_1$ . Let me just write down the ways associated with this network.

**Negation:**



$$h_\Theta(x) = g(10 - 20x_1)$$

We have only one input feature  $x_1$  in this case and the bias unit +1. And if I associate this with the weights plus 10 and -20, then my hypothesis is computing this  $h(x)$  equals sigmoid  $g(10 - 20x_1)$ . So when  $x_1$  is equal to 0, my hypothesis would be computing  $g(10 - 20 \times 0)$  is just 10. And so that's approximately 1, and when  $x_1$  is equal to 1, this will be  $g(-10)$  which is therefore approximately equal to 0. And if you look at what these values are, that's essentially the NOT  $x_1$  function. Cells include negations, the general idea is to put large negative weight in front of the variable you want to negate. Minus 20 multiplied by  $x_1$  and that's the general idea of how you end up negating  $x_1$ .

$$\rightarrow (\text{NOT } x_1) \text{ AND } (\text{NOT } x_2)$$

$\underbrace{\quad}_{\leq 1} \text{ if and only if}$

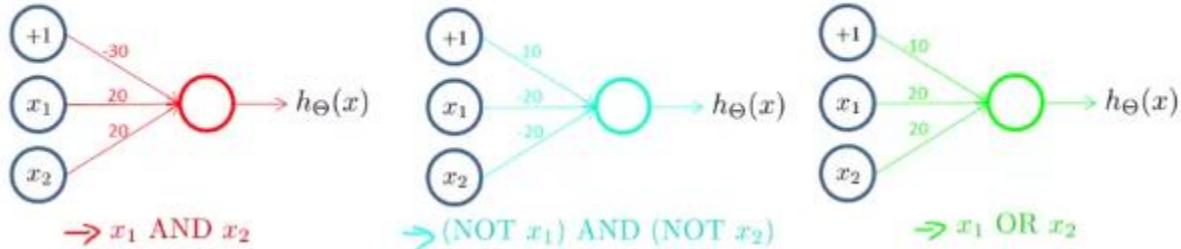
$$\rightarrow x_1 = x_2 = 0$$

And so in an example that I hope that you can figure out yourself, if you want to compute a function like this NOT  $x_1$  AND NOT  $x_2$ , part of that will probably be putting large negative weights in front of  $x_1$  and  $x_2$ , but it should be feasible. So you get a neural network with just one output unit to compute this as well. All right, so this logical function, NOT  $x_1$  AND NOT  $x_2$  is going to be equal to 1 if and only if  $x_1$  equals  $x_2$  equals 0. All right since this is a logical function, this says NOT  $x_1$  means  $x_1$  must be 0 and NOT  $x_2$  that means  $x_2$  must be equal to 0 as well. So this logical function is equal to 1 if and only if both  $x_1$  and  $x_2$  are equal to 0 and

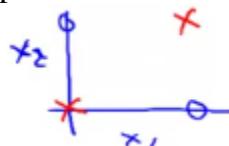
hopefully you should be able to figure out how to make a small neural network to compute this logical function as well.

Now, **taking the three pieces that we have put together** as the network for computing  $x_1 \text{ AND } x_2$ , and the network for computing  $\text{NOT } x_1 \text{ AND NOT } x_2$ , and one last network for computing  $x_1 \text{ OR } x_2$ , we should be able to put these three pieces together to compute this  $x_1 \text{ XNOR } x_2$  function.

### Putting it together: $x_1 \text{ XNOR } x_2$

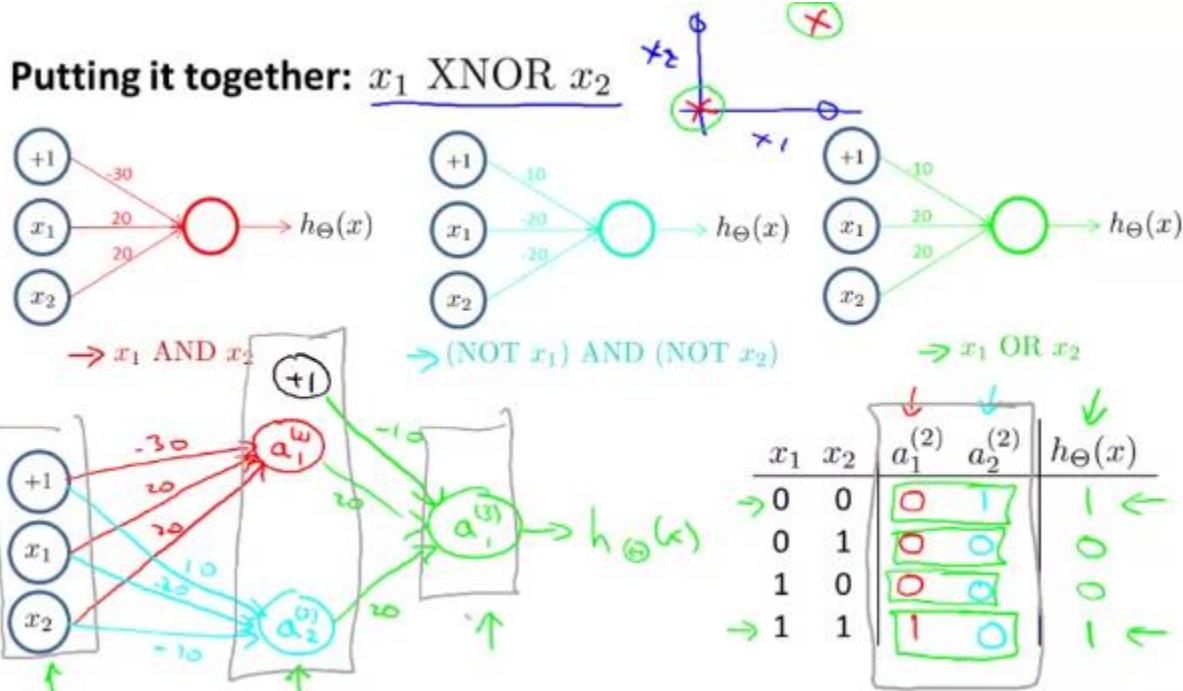


And just to remind you if this is  $x_1, x_2$ , this function that we want to compute would have negative examples here and here, and we'd have positive examples there and there.



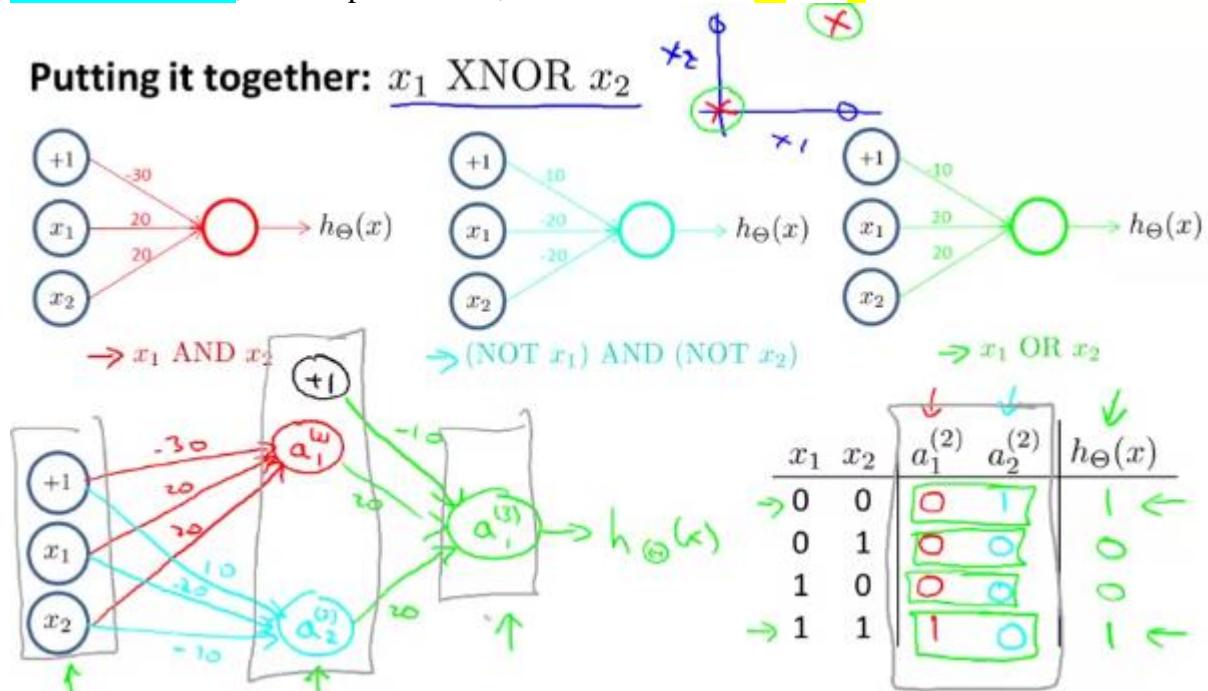
And so clearly this will need a non linear decision boundary in order to separate the positive and negative examples.

Let's draw the network.



I'm going to take my inputs  $+1, x_1, x_2$ , and create my first hidden unit here. I'm gonna call this  $a^{(2)}_1$  because that's my first hidden unit. And I'm gonna copy the **weight** over from the red network, the  $x_1$  and  $x_2$  as well, so then -30, 20, 20.

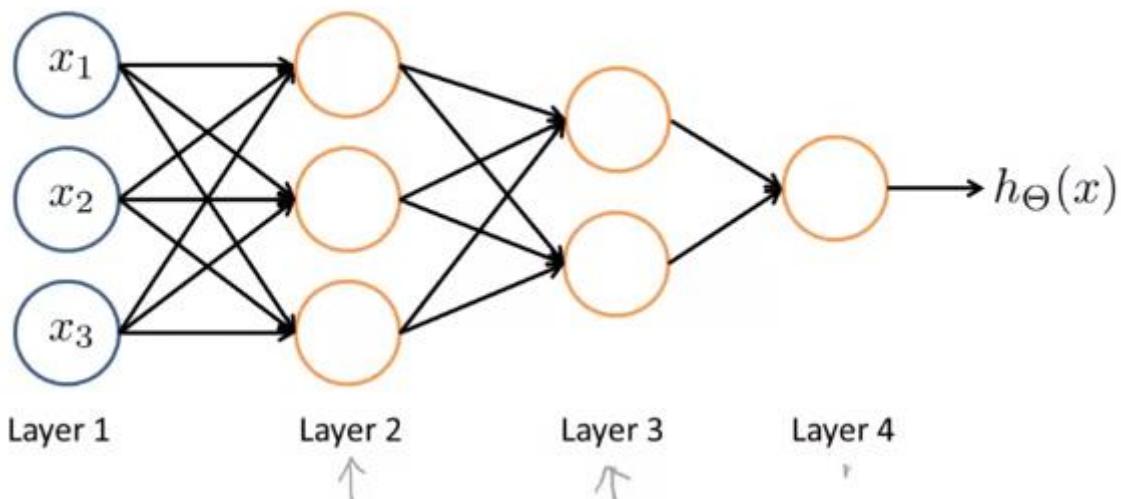
Next let me create a second hidden unit which I'm going to call  $a^{(2)}_2$ . That is the second hidden unit of layer two. I'm going to copy over the cyan network in the middle, so I'm gonna have the weights 10 -20 -20. And so, let's pull some of the truth table values. For the **red network**, we know that was computing the  $x_1 \text{ AND } x_2$ , and so this will be approximately 0 0 0 1, depending on the values of  $x_1$  and  $x_2$ , and for  $a^{(2)}_2$ , the **cyan network**, what do we know, the function **NOT  $x_1 \text{ AND NOT } x_2$** , that outputs 1 0 0 0, for the 4 values of  $x_1$  and  $x_2$ .



Finally, I'm going to create my **output node, my output unit** that is  $a^{(3)}_1$ . This is one more output  $h(x)$  and I'm going to copy over the **green network** for that. And I'm going to need a **+1 bias unit** here, so you draw that in, and I'm going to **copy over the weights from the green networks**, so that's **-10, 20, 20** and we know earlier that **this computes the OR function**. So let's fill in the truth table entries. So the first entry is 0 OR 1 which can be 1 that makes 0 OR 0 which is 0, 0 OR 0 which is 0, 1 OR 0 and that falls to 1. And thus  $h(x)$  is equal to 1 when either both  $x_1$  and  $x_2$  are 0 or when  $x_1$  and  $x_2$  are both 1 and concretely  $h(x)$  outputs 1 exactly at these two locations and then outputs 0 otherwise. And thus will this neural network, which has a input layer, one hidden layer, and one output layer, **we end up with a nonlinear decision boundary that computes this XNOR function**. And the more general intuition is that in the input layer, we just have our four inputs. Then we have a hidden layer, which computed some slightly more complex functions of the inputs that is shown here, this is slightly more complex functions. And then by adding yet another layer we end up with an even more complex non linear function.

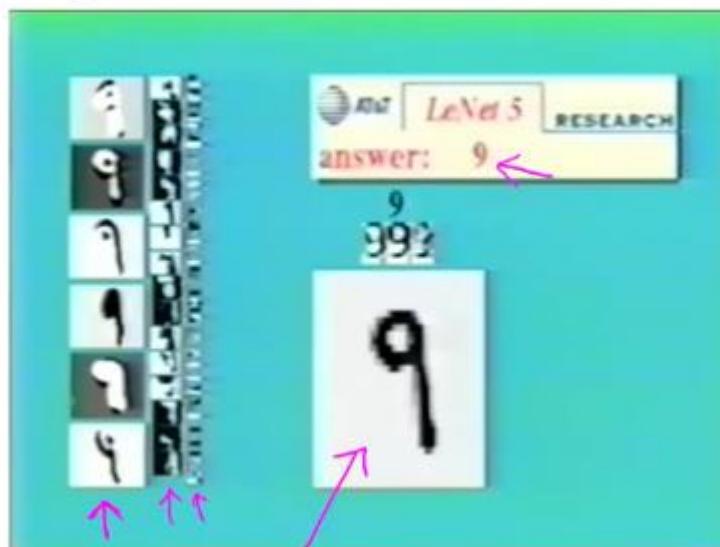
And this is a sort of intuition about why neural networks can compute pretty complicated functions that when you have multiple layers you have relatively simple function of the inputs of the second layer but the third layer **can build on that to compute even more complex functions, and then the layer after that can compute even more complex functions.**

### Neural Network intuition



To wrap up this video, I want to show you a fun example of an application of a Neural Network that captures this intuition of the deeper layers computing more complex features. I want to show you a video that a good friend of mine Yann LeCun, Yann is a professor at New York University, NYU and he was one of the early pioneers of Neural Network research and is sort of a legend in the field now and his ideas are used in all sorts of products and applications throughout the world now. So I wanna show you a video from some of his early work in which he was using a neural network to recognize handwriting, to do handwritten digit recognition. You might remember early in this class, at the start of this class I said that one of the earliest successes of neural networks was trying to use it to read zip codes to help USPS help us send mail in lots, to read postal codes.

### Handwritten digit classification



So this is one of the attempts, this is one of the algorithms used to try to address that problem. In the video that I'll show you this area here is the input area that shows a canvassing character shown to the network. This column here shows a visualization of the features computed by sort of the first hidden layer of the network. So the first hidden layer you know this visualization shows different features. Different edges and lines and so on detected. This is a visualization of the next hidden layer. It's kinda harder to see, harder to understand the deeper, hidden layers, and that's a visualization of what the next hidden layer is computing. You probably have a hard time seeing what's going on much beyond the first hidden layer, but then finally, all of these learned features get fed to the upper layer and shown over here is the final answer, it's the final predictive value for what handwritten digit the neural network thinks it is being shown.

So let's take a look at the video.

[MUSIC and VIDEO]

So I hope you enjoyed the video and that this hopefully gave you some intuition about the source of pretty complicated functions neural networks can learn. In which it takes its input this image, just takes this input, the raw pixels and the first hidden layer computes some set of features. The next hidden layer computes even more complex features and even more complex features. And these features can then be used by essentially the final layer of logistic regression classifiers to make accurate predictions about what are the numbers that the network sees.

## Summary

The  $\Theta^{(1)}$  matrices for AND, NOR, and OR are:

*AND:*

$$\Theta^{(1)} = [-30 \quad 20 \quad 20]$$

*NOR:*

$$\Theta^{(1)} = [10 \quad -20 \quad -20]$$

*OR:*

$$\Theta^{(1)} = [-10 \quad 20 \quad 20]$$

We can combine these to get the XNOR logical operator (which gives 1 if  $x_1$  and  $x_2$  are both 0 or both 1).

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \end{bmatrix} \rightarrow [a^{(3)}] \rightarrow h_{\Theta}(x)$$

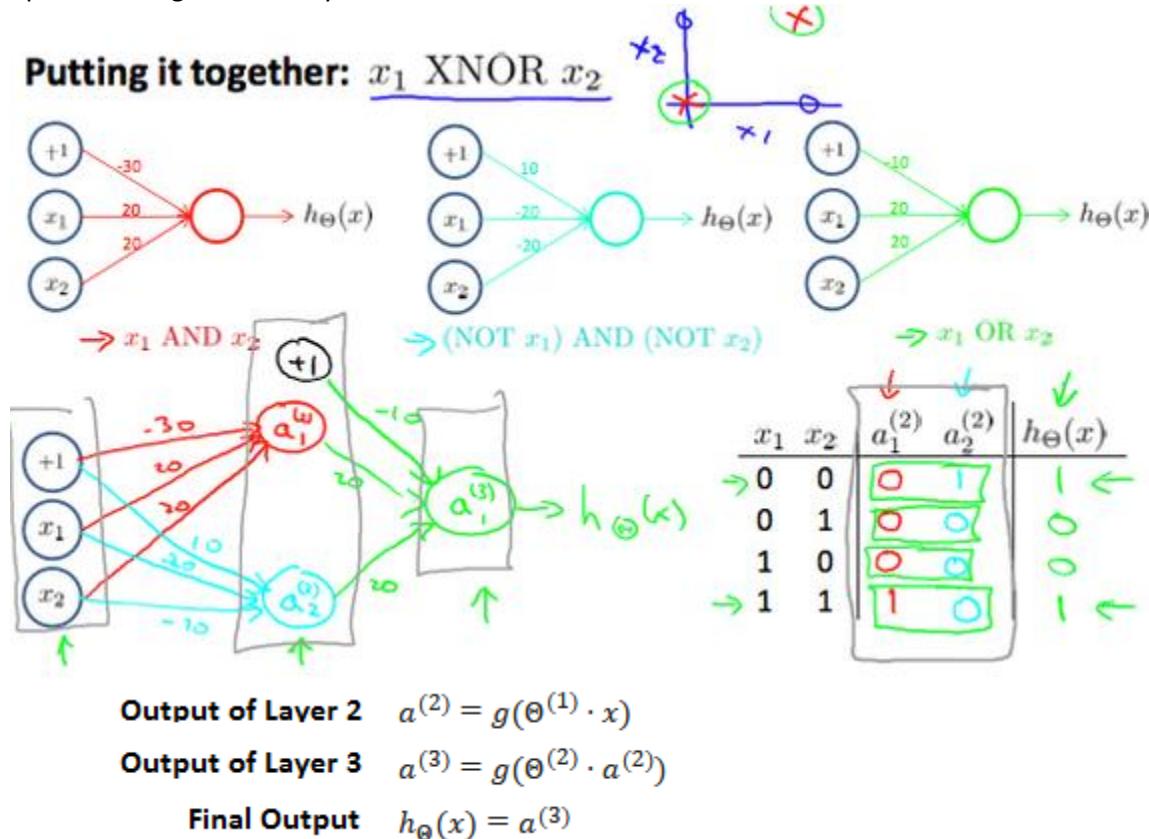
For the transition between the first and second layer, we'll use a  $\Theta^{(1)}$  matrix that combines the values for AND and NOR,

$$\Theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 \\ 10 & -20 & -20 \end{bmatrix}$$

For the transition between the second and third layer, we'll use a  $\Theta^{(2)}$  matrix that uses the value for OR:

$$\Theta^{(2)} = [-10 \quad 20 \quad 20]$$

Putting it all together, let's write out the values for all our nodes, and finally there we have the XNOR operator using a hidden layer with two nodes!

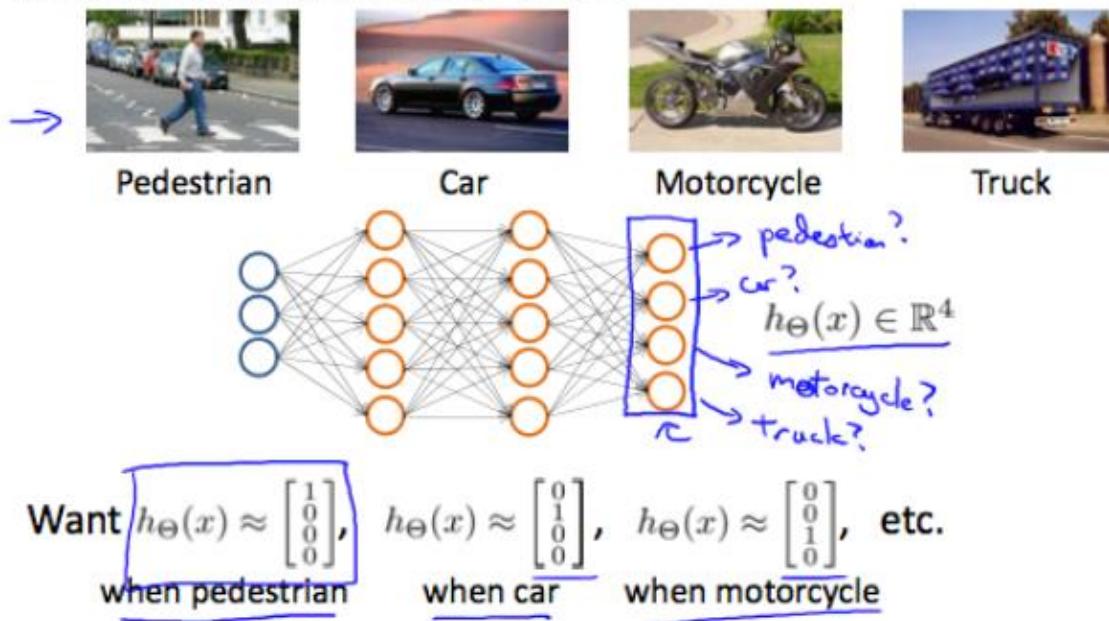


## Multiclass Classification

In this video, I want to tell you about how to use neural networks to do multiclass classification where we may have more than one category that we're trying to distinguish amongst.

In the last part of the last video, where we had the handwritten digit recognition problem, that was actually a multiclass classification problem because there were ten possible categories for recognizing the digits from 0 through 9 and so, if you want us to fill you in on the details of how to do that.

## Multiple output units: One-vs-all.

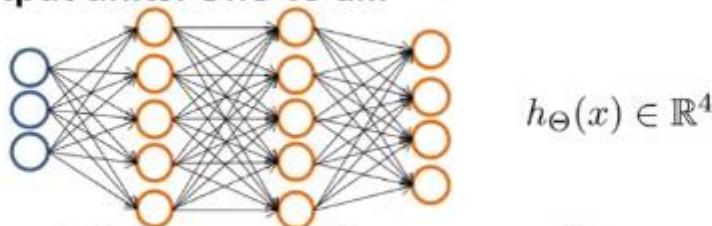


The way we do multiclass classification in a neural network is essentially **an extension of the one versus all method**. So, let's say that we have a computer vision example, where instead of just trying to recognize cars as in the original example that I started off with, but let's say that we're trying to recognize, you know, four categories of objects and given an image we want to decide if it is a pedestrian, a car, a motorcycle or a truck. If that's the case, what we would do is **we would build a neural network with four output units** so that our neural network now **outputs a vector of four numbers**. So, the output now is actually needing to be a vector of four numbers and what we're going to try to do is **get the first output unit to classify: is the image a pedestrian, yes or no**. The **second unit to classify**: is the image a **car**, yes or no. This unit to classify: is the image a **motorcycle**, yes or no, and this would classify: is the image a **truck**, yes or no.

And thus, when the image is of a **pedestrian**, we would ideally want the **network to output  $1, 0, 0, 0$** , when it is a car we want it to output  $0, 1, 0, 0$ , when this is a motorcycle, we get it to or rather, we want it to output  $0, 0, 1, 0$  and so on. So this is just like the "**one versus all**" method that we talked about when we were describing logistic regression, and **here we have essentially four logistic regression classifiers**, each of which is trying to recognize one of the four classes that we want to distinguish amongst.

So, rearranging the slide of it.

## Multiple output units: One-vs-all.



Want  $h_{\Theta}(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ ,  $h_{\Theta}(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ ,  $h_{\Theta}(x) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ , etc.  
 when pedestrian      when car      when motorcycle

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

$\rightarrow y^{(i)}$  one of  $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ ,  $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ ,  $\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ ,  $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$   
 pedestrian    car    motorcycle    truck

~~Previously~~  
 ~~$y \in \{1, 2, 3, 4\}$~~   
 ~~$h_{\Theta}(x^{(i)}) \approx y^{(i)}$~~   
 ~~$\therefore \in \mathbb{R}^4$~~

Here's our **neural network with four output** units and those are what we want  $h(x)$  to be when we have the different images, and the way we're going to represent the training set in these settings is as follows. So, when we have a training set with different images of pedestrians, cars, motorcycles and trucks, what we're going to do in this example is that whereas previously we had written out the labels as  $y$  being an integer from 1, 2, 3 or 4. Instead of representing  $y$  this way, we're going to instead represent  $y$  as follows: namely  $y_i$  will be either 1, 0, 0, 0 or 0, 1, 0, 0 or 0, 0, 1, 0 or 0, 0, 0, 1 depending on what the corresponding image  $x_i$  is. And so one training example will be one pair  $(x_i, y_i)$  where  $x_i$  is an image with, you know one of the four objects and  $y_i$  will be one of these vectors.

And hopefully, we can find a way to get our Neural Networks to output some value. So, the  $h(x)$  is approximately  $y$  and both  $h(x)$  and  $y_i$ , both of these are going to be in our example, four dimensional vectors when we have four classes.

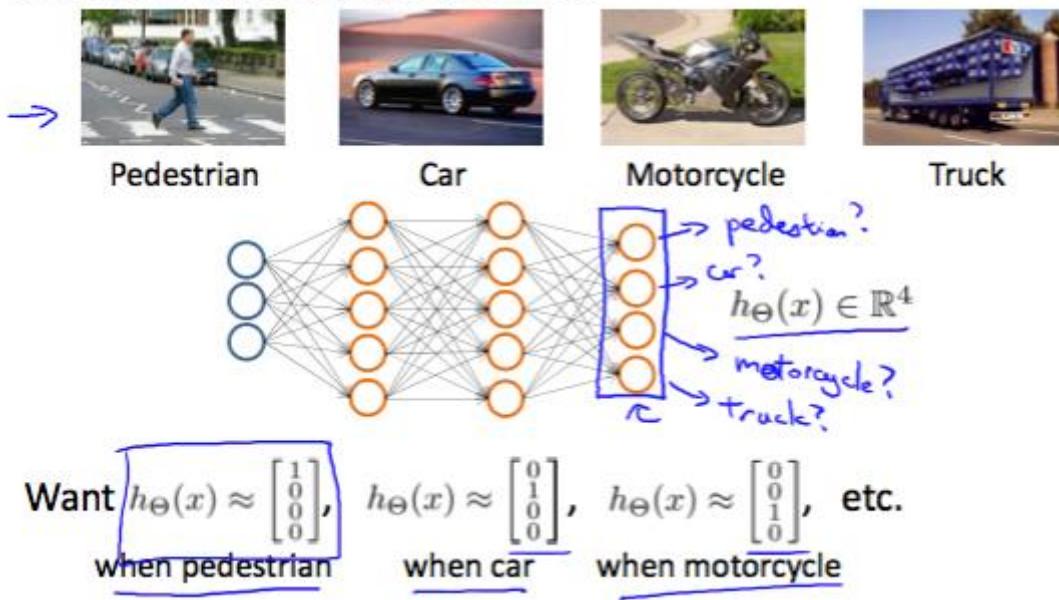
So, that's how you get neural network to do multiclass classification. This wraps up our discussion on how to represent Neural Networks that is on our hypotheses representation.

In the next set of videos, let's start to talk about **how take a training set and how to automatically learn the parameters of the neural network.**

## Summary - Multiclass Classification

To classify data into multiple classes, we **let our hypothesis function return a vector of values**. Say we wanted to classify our data into one of four categories. We will use the following example to see how this classification is done. This algorithm takes as input an image and classifies it accordingly:

### Multiple output units: One-vs-all.



We can define our set of resulting classes as  $y$ :

$$y^{(i)} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix},$$

Each  $y^{(i)}$  represents a different image corresponding to either a car, pedestrian, truck, or motorcycle. The inner layers, each provide us with some new information which leads to our final hypothesis function. The setup looks like:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} \rightarrow \begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \\ \dots \end{bmatrix} \rightarrow \begin{bmatrix} a_0^{(3)} \\ a_1^{(3)} \\ a_2^{(3)} \\ \dots \end{bmatrix} \rightarrow \dots \rightarrow \begin{bmatrix} h_{\Theta}(x)_1 \\ h_{\Theta}(x)_2 \\ h_{\Theta}(x)_3 \\ h_{\Theta}(x)_4 \end{bmatrix}$$

Our resulting hypothesis for one set of inputs may look like:

$$h_{\Theta}(x) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

In which case our resulting class is the third one down, or  $h_{\Theta}(x)_3$ , which represents the motorcycle.

# Neural Network - Cost Function

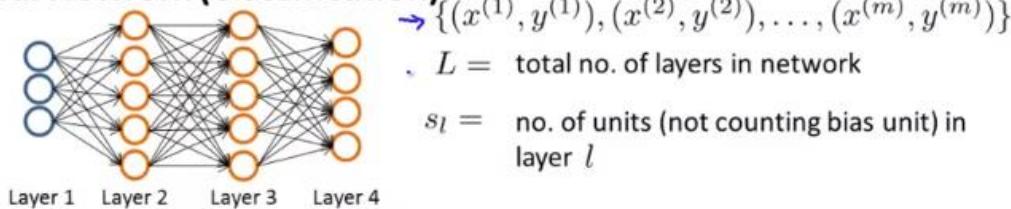
Neural networks are one of the most powerful learning algorithms that we have today.

In this and in the next few videos, I'd like to start talking about **a learning algorithm** for **fitting the parameters of a neural network** given a **training set**.

As with the discussion of most of our learning algorithms, we're going to begin by talking about the cost function for fitting the parameters of the network. I'm going to focus on the **application of neural networks** to **classification problems**.

So suppose we have a **network like that** shown on the left. And suppose we have a training set like this is  $(x_i, y_i)$  pairs of  $m$  training example.

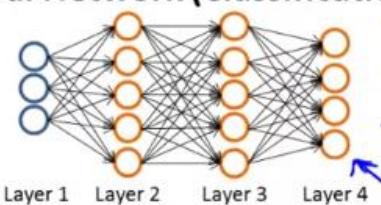
## Neural Network (Classification)



I'm going to use **upper case L** to denote the total **number of layers** in this network. So for the network shown on the left we would have capital L equals 4. I'm going to use  $s_l$  to denote the **number of units** that is the **number of neurons**, not counting the bias unit in layer 1 of the network. So for example, we would have  $s_1$ , which is equal there, equals 3 units,  $s_2$  in my example is 5 units. And the output layer  $s_4$ , which is also equal to  $s_L$  because capital L is equal to 4, the output layer in my example under that has four units.

We're going to consider **two types of classification problems**.

The first is **Binary classification**, where the labels  $y$  are either **0 or 1**. In this case, we will have 1 output unit, so this Neural Network unit on top has 4 output units, but if we had binary classification **we would have only one output unit** that computes  $h(x)$ .

**Neural Network (Classification)**

$\rightarrow \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$

$\rightarrow L = \text{total no. of layers in network}$   $L=4$

$\rightarrow s_l = \text{no. of units (not counting bias unit) in layer } l$   $s_1=3, s_2=5, s_3=s_4=L=4$

**Binary classification**

$y = 0 \text{ or } 1$

$$h_\theta(x)$$

1 output unit

$$h_\theta(x) \in \mathbb{R}$$

$$s_L=1, \quad K=1$$

**Multi-class classification (K classes)**

$y \in \mathbb{R}^K$  E.g.  $\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$  ←  
pedestrian car motorcycle truck

K output units

$$h_\theta(x) \in \mathbb{R}^k$$

$$s_L = K \quad (K \geq 3)$$

And the output of the neural network would be **h(x) is going to be a real number**. And in this case the **number of output units**,  $s_L$ , where  $L$  is again the index of the final layer. Cuz that's the number of layers we have in the network so the **number of units we have in the output layer** is going to be equal to **1**.

In this case to simplify notation later, I'm also going to set  $K=1$  so you can **think of K as also denoting the number of units in the output layer**.

The **second type of classification problem** we'll consider will be **multi-class classification** problem where we may have **K distinct classes**. So our earlier example had this representation for  $y$  if we have 4 classes, and in this case we will have **capital K output units** and **our hypothesis or output vectors that are K dimensional**. And the number of output units will be equal to  $K$ . And usually we would have **K greater than or equal to 3** in this case, because **if we had two cases, then we don't need to use the one versus all method**. We **use the one versus all method only if we have K greater than or equals 3 classes**, so having only two classes we will need to use only one upper unit.

Now **let's define the cost function** for our neural network.

The cost function we use for the neural network is going to be a **generalization** of the one that we use for **logistic regression**. For logistic regression we used to minimize the cost function  $J(\theta)$  that was **minus 1/m of this cost function** and then **plus this extra regularization term** here, where this was a sum from  $j=1$  through  $n$ , because we did not regularize the bias term  $\theta_0$ .

**Cost function****Logistic regression:**

$$J(\theta) = -\frac{1}{m} \left[ \sum_{i=1}^m y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

$\Theta_0$

For a neural network, our cost function is going to be a generalization of this. Where instead of having basically just one, which is the classification output unit, we may instead have K of them.

So here's our cost function.

### Cost function

Logistic regression:

$$J(\theta) = -\frac{1}{m} \left[ \sum_{i=1}^m y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

Neural network:

$$\rightarrow h_\Theta(x) \in \mathbb{R}^K \quad (h_\Theta(x))_i = i^{th} \text{ output}$$

$$\rightarrow J(\Theta) = -\frac{1}{m} \left[ \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k) \right]$$

$$\boxed{\frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{ji}^{(l)})^2}$$

Our neural network now outputs vectors in  $\mathbb{R}^K$  where K might be equal to 1 if we have a binary classification problem.

I'm going to use this notation  $h(\mathbf{x})_i$  to denote the  $i^{th}$  output. That is,  $h(\mathbf{x})$  is a  $K$ -dimensional vector and so this subscript  $i$  just selects out the  $i^{th}$  element of the vector that is output by my neural network.

My cost function  $J(\theta)$  is now going to be the following. Minus 1 over m of a sum of a similar term to what we have for logistic regression, except that we have the sum from k equals 1 through K. This summation is basically a sum over my K output units.

So if I have 4 output units, that is if the final layer of my neural network has 4 output units, then this is a sum from k equals 1 through 4 of basically the logistic regression algorithm's cost function but summing that cost function over each of my four output units in turn. And so you notice in particular that this applies to  $y_k h_k$ , because we're basically taking the K upper units, and comparing that to the value of  $y_k$ , which is that one of those vectors saying what cost it should be.

And finally, the second term here is the regularization term, similar to what we had for logistic regression. This summation term looks really complicated, but all it's doing is it's summing over these terms  $\Theta_{ji}^{(l)}$  for all values of  $i$ ,  $j$  and  $l$ .

Except that we don't sum over the terms corresponding to these bias values like we have for logistic progression. Concretely, we don't sum over the terms corresponding to where  $i$  is equal to 0. So that is because when we're computing the activation of a neuron, we have terms like

these,  $\theta_{i0} x_0 + \theta_{i1} x_1 + \dots$  and so on, where I guess put in a two there, this is the first in there. And so the values with a zero there, that corresponds to something that multiplies into an  $x_0$  or an  $a_0$  and so this is kinda like a bias unit and by analogy to what we were doing for logistic progression, we won't sum over those terms in our regularization term because we don't want to regularize them and string their values as zero.

But this is just one possible convention, and even if you were to sum over  $i$  equals 0 up to  $S_l$ , it would work about the same and doesn't make a big difference. But maybe this **convention of not regularizing the bias term is just slightly more common**. So **that's the cost function we're going to use for our neural network**.

In the **next video** we'll start to talk about **an algorithm for trying to optimize the cost function**.

## Summary – Neural Network Cost Function

Let's first define a few variables that we will need to use:

- $L$  = total number of layers in the network
- $S_l$  = number of units (not counting bias unit) in layer  $l$
- $K$  = number of output units/classes

Recall that in neural networks, we may have many output nodes. We denote  $h_\theta(x)_k$  as being a hypothesis that results in the  $k^{\text{th}}$  output. Our **cost function for neural networks** is going to be a **generalization of the one we used for logistic regression**.

Recall that the **cost function for regularized logistic regression was**:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

**For neural networks**, it is going to be slightly more complicated:

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K \left[ y_k^{(i)} \log((h_\Theta(x^{(i)}))_k) + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k) \right] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{S_l} \sum_{j=1}^{S_{l+1}} (\Theta_{j,i}^{(l)})^2$$

We have **added a few nested summations to account for our multiple output nodes**.

**In the first part** of the equation, before the square brackets, we have an additional nested summation that loops through the number of output nodes.

**In the regularization part**, after the square brackets, we must account for **multiple  $\theta$  matrices**.

- The **number of columns in our current  $\theta$  matrix** is equal to the **number of nodes in our current layer (including the bias unit)**.

- The number of rows in our current  $\theta$  matrix is equal to the number of nodes in the next layer (excluding the bias unit).

As before with logistic regression, we square every term.

Note:

- the double sum simply adds up the logistic regression costs calculated for each cell in the output layer
- the triple sum simply adds up the squares of all the individual  $\Theta$ s in the entire network.
- the  $i$  in the triple sum does not refer to training example  $i$

## Back propagation Algorithm

In the previous video, we talked about a cost function for the neural network.

In this video, let's start to talk about an algorithm, for trying to minimize the cost function. In particular, we'll talk about the back propagation algorithm.

Here's the cost function that we wrote down in the previous video.

### Gradient computation

$$\begin{aligned} \Rightarrow J(\Theta) = & -\frac{1}{m} \left[ \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log h_\theta(x^{(i)})_k + (1 - y_k^{(i)}) \log(1 - h_\theta(x^{(i)})_k) \right] \\ & + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_j^{(l)})^2 \end{aligned}$$

$$\Rightarrow \min_{\Theta} J(\Theta)$$

Need code to compute:

$$\begin{aligned} \rightarrow -J(\Theta) \\ \rightarrow -\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) \end{aligned}$$

$$\Theta_{ij}^{(l)} \in \mathbb{R}$$

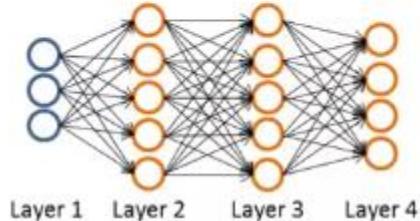
What we'd like to do is try to find parameters  $\theta$  to try to minimize  $J(\theta)$ .

In order to use either gradient descent or one of the advance optimization algorithms what we need to do therefore is to write code that takes this input the parameters  $\theta$  and computes  $J(\theta)$  and these partial derivative terms. Remember, that the parameters in the neural network of these things,  $\theta_{ij}^{(l)}$ , that's the real number and so, these are the partial derivative terms we need to compute.

In order to compute the cost function  $J(\theta)$ , we just use this formula up here and so, what I want to do for the most of this video is focus on talking about **how we can compute these partial derivative terms**.

## Gradient computation

Given one training example ( $x, y$ ):



Let's start by talking about the case of when we have only one training example, so imagine, if you will, that our entire training set comprises **only one training example** which is a **pair**  $(x, y)$ . I'm not going to write  $x_1 y_1$ , just write this, write a one training example as  $(x, y)$  and **let's step through the sequence of calculations** we would do with this one training example.

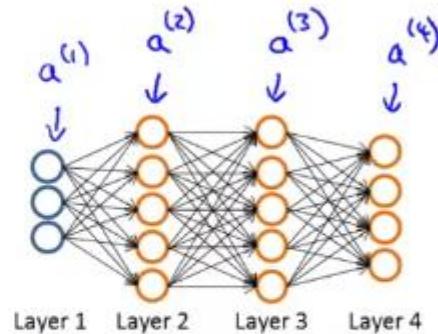
The first thing we do is we apply **forward propagation** in order to **compute what a hypotheses actually outputs given this input x**.

## Gradient computation

Given one training example ( $x, y$ ):

Forward propagation:

$$\begin{aligned} \underline{a^{(1)}} &= \underline{x} \\ \rightarrow z^{(2)} &= \Theta^{(1)} a^{(1)} \\ \rightarrow a^{(2)} &= g(z^{(2)}) \quad (\text{add } a_0^{(2)}) \\ \rightarrow z^{(3)} &= \Theta^{(2)} a^{(2)} \\ \rightarrow a^{(3)} &= g(z^{(3)}) \quad (\text{add } a_0^{(3)}) \\ \rightarrow z^{(4)} &= \Theta^{(3)} a^{(3)} \\ \rightarrow a^{(4)} &= h_{\Theta}(x) = g(z^{(4)}) \end{aligned}$$



Concretely, remember that we called  $\underline{a^{(1)}}$  is the activation values of this first layer that was the input layer, so we set that equal to  $x$ . And then we're going to compute  $\underline{z^{(2)}}$  equals  $\underline{\Theta^{(1)}a^{(1)}}$  and  $\underline{a^{(2)}}$  equals  $\underline{g}$ , the sigmoid activation function applied to  $\underline{z^{(2)}}$  and this would give us our activations for the first hidden layer, that is for layer two of the network, and we also add those **bias terms**. Next we apply 2 more steps of this forward propagation to compute  $\underline{a^{(3)}}$  and  $\underline{a^{(4)}}$  which is also the upwards of a hypotheses  $h$  of  $x$ . So this is our **vectorized implementation of forward propagation and it allows us to compute the activation values for all of the neurons in our neural network**.

**Next**, in order to compute the derivatives, we're going to use an algorithm called **back propagation**. The intuition of the back propagation algorithm is that for each node we're going to compute the term  $\delta^{(l)}_j$  that's going to somehow represent the error of node j in the layer l.

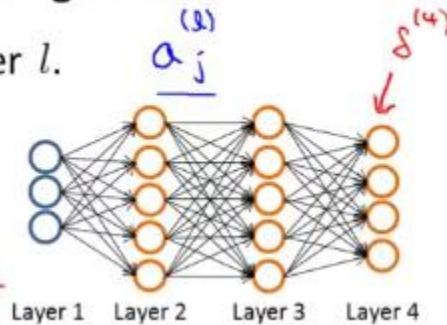
So, recall that  $a^{(l)}_j$  that does the activation of the j<sup>th</sup> unit in layer l, and so this **delta term** is in some sense going to capture our error in the activation of that node, so, how we might wish the activation of that node was slightly different.

### Gradient computation: Backpropagation algorithm

Intuition:  $\delta_j^{(l)}$  = "error" of node j in layer l.

For each output unit (layer L = 4)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$



Concretely, taking the example neural network that we have on the right, which has four layers, and so **capital L is equal to 4**. For each output unit, we're going to compute this  $\delta$  term.

So,  **$\delta$  for the j<sup>th</sup> unit** in the fourth layer is equal to just the **activation of that unit minus** what was the **actual value of j<sup>th</sup> in our training example**. So, this term here can also be written  $(h(x))_j$ , right, so this  $\delta$  term is just the difference between when a hypotheses output and what was the value of y in our training set, whereas  $y_j$  is the **j<sup>th</sup> element of the vector value y in our labeled training set**.

And by the way, if you think of  $\delta$ ,  $a$ , and  $y$  as vectors, then you can also take those and come up with a **vectorized implementation** of it, which is just  $\delta^{(4)}$  gets set as  $a^{(4)}$  minus  $y$ . Where here, each of these  $\delta^{(4)}$ ,  $a^{(4)}$ , and  $y$ , each of these is a vector whose dimension is equal to the number of output units in our network. So we've now computed the error term's  $\delta^{(4)}$  for our network.

What we do **next** is compute the **delta terms for the earlier layers** in our network.

$$\begin{aligned} \delta^{(3)} &= (\Theta^{(3)})^T \delta^{(4)} \cdot g'(z^{(3)}) \\ \delta^{(2)} &= (\Theta^{(2)})^T \delta^{(3)} \cdot g'(z^{(2)}) \end{aligned}$$

(No  $\delta^{(1)}$ )

Here's a formula for computing  $\delta^{(3)}$ .  $\delta^{(3)}$  is equal to  $(\Theta^{(3)})^T \delta^{(4)}$ . And this dot times, this is the element wise multiplication operation that we know from MATLAB. So  $(\Theta^{(3)})^T \delta^{(4)}$ , that's a **vector**;  $g'(z^{(3)})$  that's also a **vector**, and so **dot times** is an **element wise multiplication** between these two vectors. This term  $g'(z^{(3)})$  that formally is actually the **derivative of the activation function g** evaluated at the input values given by  $z^{(3)}$ .

If you know calculus, you can try to work it out yourself and see that you can simplify it to the same answer that I get. But I'll just tell you pragmatically what that means. What you do to

compute this  $\mathbf{g}'$ , these derivative terms is just  $\mathbf{a}^{(3)}$  dot times 1 minus  $\mathbf{a}^{(3)}$  where  $\mathbf{a}^{(3)}$  is the vector of activations, 1 is the vector of ones, and  $\mathbf{a}^{(3)}$  is again the activation, the **vector of activation values for that layer**. Next you apply a similar formula to compute  $\delta^{(2)}$  where again that can be computed using a similar formula. Only now it is  $\mathbf{a}^{(2)}$  like so and I then prove it here but you can actually, it's possible to prove it if you know calculus that this expression is equal to mathematically, **the derivative of the g function**, of the activation function, which I'm denoting by  $\mathbf{g}'$ .

And finally, that's it and there is no  $\delta^{(1)}$  term, because the first layer corresponds to the input layer and that's just the features we observed in our training set, so that doesn't have any error associated with it. It's not like, you know, we don't really want to try to change those values. And so we have delta terms only for layers 2, 3 and 4 in this example. The name back propagation comes from the fact that we start by computing the delta term for the output layer and then we go back a layer and compute the delta terms for the third hidden layer and then we go back another step to compute delta 2 and so, we're sort of back propagating the errors from the output layer to layer 3 to layer 2, hence the name back complication.

Finally, the derivation is surprisingly complicated, surprisingly involved, but if you just do this few steps of computation it is possible to prove, very frankly, some what complicated mathematical proof, **it's possible to prove** that if you ignore regularization then the **partial derivative terms** you want are **exactly given by the activations and these delta terms**. This is ignoring lambda or alternatively the regularization term lambda will equal to 0.

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = \alpha_j^{(l)} \delta_i^{(l+1)} \quad (\text{ignoring } \lambda; \text{ if } \lambda = 0) \leftarrow$$

We'll fix this detail later about the regularization term, but so by performing back propagation and computing these delta terms, you can, you know, pretty quickly compute these partial derivative terms for all of your parameters.

So this is a lot of detail. Let's take everything and put it all together to **talk about how to implement back propagation to compute derivatives with respect to your parameters**. And for the case of when we have a large training set, not just a training set of one example, here's what we do.

Suppose we have a training set of  $m$  examples like that shown here. The first thing we're going to do is we're going to set these  $\Delta_{ij}^{(l)}$ . So this triangular symbol, that's actually the capital Greek alphabet delta. The symbol we had on the previous slide was the lower case  $\delta$  delta. So the triangle is capital delta. We're gonna set this equal to zero for all values of  $i, j$ .

### Backpropagation algorithm

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

Set  $\Delta_{ij}^{(l)} = 0$  (for all  $l, i, j$ ). *(use to compute  $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$ )*

Eventually, this  $\Delta_{ij}^{(l)}$  will be used to compute the **partial derivative term**, partial derivative with respect to  $\theta_{ij}^{(l)}$  of  $J(\theta)$ . So as we'll see in a second, **these deltas are going to be used as accumulators that will slowly add things in order to compute these partial derivatives.**

Next, we're going to loop through our training set.

For  $i = 1$  to  $m \leftarrow (\underline{x}^{(i)}, \underline{y}^{(i)})$ .

Set  $a^{(1)} = \underline{x}^{(i)}$

- Perform forward propagation to compute  $a^{(l)}$  for  $l = 2, 3, \dots, L$
- Using  $\underline{y}^{(i)}$ , compute  $\delta^{(L)} = \underline{a}^{(L)} - \underline{y}^{(i)}$
- Compute  $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$
- $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

So, we'll say for  $i$  equals 1 through  $m$ , and so for the  $i^{\text{th}}$  iteration, we're going to be working with the training example  $(\underline{x}_i, \underline{y}_i)$ . So the first thing we're going to do is set  $\underline{a}^{(1)}$  which is the activations of the input layer, set that to be equal to  $\underline{x}^{(i)}$  that is the inputs for our  $i^{\text{th}}$  training example, and then we're going to perform forward propagation to compute the activations for layer two, layer three and so on up to the final layer, layer capital  $L$ . Next, we're going to use the output label  $\underline{y}_i$  from this specific example we're looking at, to compute the error term for  $\delta^{(L)}$  for the output there. So  $\delta^{(L)}$  is what a **hypotheses output minus** what the **target label** was?

And then we're going to use the back propagation algorithm to compute  $\delta^{(L-1)}, \delta^{(L-2)},$  and so on down to  $\delta^{(2)}$  and once again **there is no  $\delta^{(1)}$**  because we don't associate an error term with the input layer.

→ Compute  $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$

→  $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

~~$\Delta^{(1)} := \Delta^{(1)} + \delta^{(1)} (\underline{a}^{(1)})^T$~~

And finally, we're going to use these capital delta terms to accumulate these partial derivative terms that we wrote down on the previous line.

And by the way, if you look at this expression, it's possible to vectorize this too. Concretely, if you think of  $\Delta_{ij}^{(l)}$  as a matrix, indexed by subscript  $ij$ . Then, if delta  $L$  is a matrix we can rewrite this as  $\Delta^{(l)}$ , gets updated as  $\Delta^{(l)} + \delta^{(L+1)}$  times  $(\underline{a}^{(L)})^T$ .

So that's a vectorized implementation of this that automatically does this update for all values of  $i$  and  $j$ . Finally, after executing the body of the for-loop we then go outside the for-loop and we compute the following.

$$\begin{aligned} \rightarrow D_{ij}^{(l)} &:= \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \rightarrow D_{ij}^{(l)} &:= \frac{1}{m} \Delta_{ij}^{(l)} & \text{if } j = 0 \end{aligned}$$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

We compute **capital D** as follows, and we have two separate cases for  $j$  equals zero and  $j$  not equals zero. The case of  $j$  equals zero corresponds to the bias term so when  $j$  equals zero that's

why we're missing this extra regularization term. Finally, while the formal proof is pretty complicated what you can show is that once you've computed these **D terms**, that is **exactly the partial derivative of the cost function with respect to each of your parameters** and so you can use those in either gradient descent or in one of the advanced authorization algorithms.

So that's the back propagation algorithm and **how you compute derivatives of your cost function for a neural network**.

I know this looks like this was a lot of details and this was a lot of steps strung together. But both in the programming assignments write out and later in this video, we'll give you a summary of this so we can have all the pieces of the algorithm together so that you know exactly what you need to implement if you want to **implement back propagation to compute the derivatives of your neural network's cost function with respect to those parameters**.

## Summary – Back Propagation Algorithm

"**Back Propagation**" is **neural-network terminology for minimizing our cost function**, just like what we were doing with **gradient descent in logistic and linear regression**. Our goal is to compute:

$$\min_{\Theta} J(\Theta)$$

That is, **we want to minimize our cost function J** using an **optimal set of parameters in theta**. In this section we'll look at the **equations we use to compute the partial derivative of J(θ)**:

$$\frac{\partial}{\partial \Theta_{i,j}^{(l)}} J(\Theta)$$

To do so, **we use the following algorithm**:

### Backpropagation algorithm

- Training set  $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$
- Set  $\Delta_{ij}^{(l)} = 0$  (for all  $l, i, j$ ). (use to compute  $\frac{\partial}{\partial \Theta_{i,j}^{(l)}} J(\Theta)$ )
- For  $i = 1$  to  $m$  ←  $(\underline{x}^{(i)}, \underline{y}^{(i)})$ .
  - Set  $a^{(1)} = \underline{x}^{(i)}$
  - Perform forward propagation to compute  $\underline{a}^{(l)}$  for  $l = 2, 3, \dots, L$
  - Using  $\underline{y}^{(i)}$ , compute  $\delta^{(L)} = \underline{a}^{(L)} - \underline{y}^{(i)}$
  - Compute  $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$   ~~$\delta^{(1)}$~~
  - $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
- $\rightarrow D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$  if  $j \neq 0$
- $\rightarrow D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)}$  if  $j = 0$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

### Back propagation Algorithm

Given training set  $\{(x^{(1)}, y^{(1)}) \dots (x^{(m)}, y^{(m)})\}$

- Set  $\Delta_{i,j}^{(l)} := 0$  for all  $(l,i,j)$ , (hence you end up having a matrix full of zeros)

For training example  $t = 1$  to  $m$ :

- Set  $a^{(1)} := x^{(t)}$
- Perform forward propagation to compute  $a^{(l)}$  for  $l = 2, 3, \dots, L$

#### Gradient computation

Given one training example  $(x, y)$ :

Forward propagation:

$$\begin{aligned} a^{(1)} &= x \\ \Rightarrow z^{(2)} &= \Theta^{(1)} a^{(1)} \\ \Rightarrow a^{(2)} &= g(z^{(2)}) \quad (\text{add } a_0^{(2)}) \\ \Rightarrow z^{(3)} &= \Theta^{(2)} a^{(2)} \\ \Rightarrow a^{(3)} &= g(z^{(3)}) \quad (\text{add } a_0^{(3)}) \\ \Rightarrow z^{(4)} &= \Theta^{(3)} a^{(3)} \\ \Rightarrow a^{(4)} &= h_{\Theta}(x) = g(z^{(4)}) \end{aligned}$$

- Using  $y^{(t)}$ , compute  $\delta^{(L)} = a^{(L)} - y^{(t)}$

- Compute

$$\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)} \text{ using } \delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) . * a^{(l)} . * (1 - a^{(l)})$$

The delta values of layer  $l$  are calculated by multiplying the delta values in the next layer with the theta matrix of layer  $l$ . We then element-wise multiply that with a function called  $g'$ , or  $g'$ , which is the derivative of the activation function  $g$  evaluated with the input values given by  $z^{(l)}$ .

The **g-prime derivative terms** can also be written out as:

$$g'(z^{(l)}) = a^{(l)} . * (1 - a^{(l)})$$

- $\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$  or with vectorization  $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

Hence we update our new  $\Delta$  matrix.

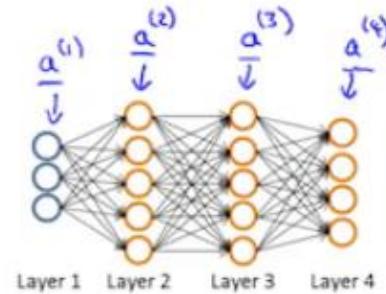
- $D_{i,j}^{(l)} := \frac{1}{m} (\Delta_{i,j}^{(l)} + \lambda \Theta_{i,j}^{(l)})$ , if  $j \neq 0$ .

- $D_{i,j}^{(l)} := \frac{1}{m} \Delta_{i,j}^{(l)}$  if  $j = 0$

The **capital-delta matrix D** is used as an "accumulator" to add up our values as we go along and eventually compute our **partial derivative**.

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

Thus we get



## Back propagation Intuition

In the previous video, we talked about the back propagation algorithm.

To a lot of people seeing it for the first time, their first impression is often that wow this is a really complicated algorithm, and there are all these different steps, and I'm not sure how they fit together. And it's kinda this black box of all these complicated steps. In case that's how you're feeling about back propagation, that's actually okay. Back propagation may be unfortunately is a less mathematically clean, or less mathematically simple algorithm, compared to linear regression or logistic regression. And I've actually used back propagation, you know, pretty successfully for many years.

And even today I still don't sometimes feel like I have a very good sense of just what it's doing, or intuition about what back propagation is doing. If, for those of you that are doing the programming exercises, that will at least mechanically step you through the different steps of how to implement back prop. So you'll be able to get it to work for yourself.

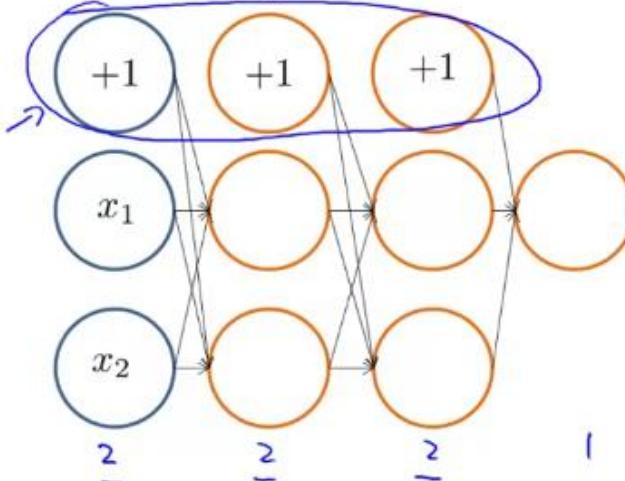
And what I want to do in this video is look **a little bit more at the mechanical steps of back propagation**, and try to give you a little more intuition about what the mechanical steps the back prop is doing to hopefully convince you that, you know, it's at least a reasonable algorithm.

In case even after this video, in case back propagation still seems very black box and kind of like a, too many complicated steps and a little bit magical to you, that's actually okay. And Even though I've used back prop for many years, sometimes this is a difficult algorithm to understand, but hopefully this video will help a little bit.

In order to better understand back propagation, let's take another closer look at what forward propagation is doing.

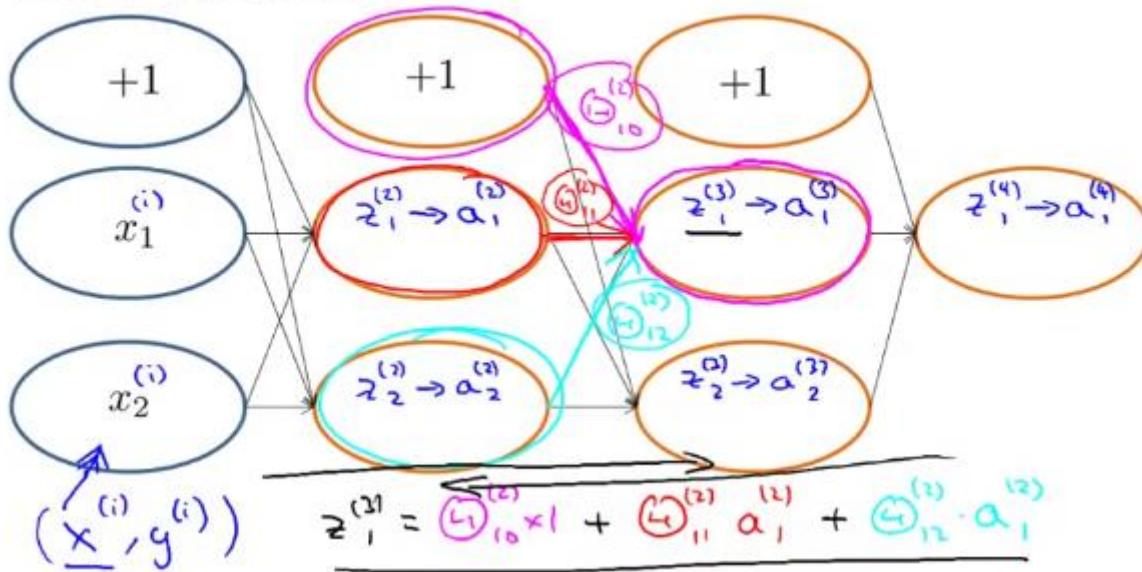
**Here's a neural network** with two input units that is not counting the bias unit, and two hidden units in this layer, and two hidden units in the next layer. And then, finally, one output unit. Again, these counts two, two, two, are not counting these bias units on top.

### Forward Propagation



In order to illustrate forward propagation, I'm going to draw this network a little bit differently. And in particular I'm going to draw this neural-network with the nodes drawn as these very fat ellipsis, so that I can write text in them.

### Forward Propagation



When performing forward propagation, we might have some particular example, say some example  $\mathbf{x}^{(i)}, \mathbf{y}^{(i)}$ . And it'll be this  $\mathbf{x}^{(i)}$  that we feed into the input layer. So this maybe  $x_1^{(i)}$  and  $x_2^{(i)}$  are the values we set the input layer to. And when we forward propagated to the first hidden layer here, what we do is compute  $z_1^{(2)}$  and  $z_2^{(2)}$ , so these are the **weighted sum of inputs of the input units**. And then we **apply the sigmoid or the logistic function**, and the **sigmoid activation function** applied to the  $z$  value. Here's are the activation values. So that gives us a  $a_1^{(2)}$  and  $a_2^{(2)}$ . And then we forward propagate again to get here  $z_1^{(3)}$ , **apply the sigmoid or the logistic function**, the activation function to that to get  $a_1^{(3)}$ . And similarly, like so until we get  $z_1^{(4)}$ , apply the activation function. This gives us  $a_1^{(4)}$ , which is the **final output value** of the neural network.

Let's erase this arrow to give myself some more space.

And if you look at what this computation really is doing, focusing on this hidden unit, let's say. We have to **add this weight**, shown in **magenta** there is my weight  $\theta^{(2)10}$ , the indexing is not important. And this way here, which I'm highlighting in **red**, that is  $\theta^{(2)11}$  and this weight here, which I'm drawing in **cyan**, is  $\theta^{(2)12}$ . So the way we compute this value,  $z^{(3)1}$  is,  $z^{(3)1}$  is as equal to this **magenta weight times this value**, so that's  $\theta^{(2)10}$  times 1, and then **plus this red weight times this value**, so that's  $\theta^{(2)11}$  times  $a^{(2)1}$ . And finally this **cyan weight times this value**, which is therefore plus  $\theta^{(2)12}$  times  $a^{(2)2}$ . And so **that's forward propagation**. And it turns out that as we'll see later in this video, **what back propagation is doing is doing a process very similar to this**. Except that instead of the computations flowing from the left to the right of this network, the computations instead flow from the right to the left of the network. And using a very similar computation as this. And I'll say in two slides exactly what I mean by that.

To better understand what back propagation is doing, let's look at the cost function. It's just the cost function that we had for when we have only one output unit. **If we have more than one output unit, we just have a summation** you know over the output units indexed by  $k$  there. If you have only one output unit then this is a cost function. And **we do forward propagation and back propagation on one example at a time**.

### What is backpropagation doing?

$$J(\Theta) = -\frac{1}{m} \left[ \sum_{i=1}^m y^{(i)} \log(h_\Theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\Theta(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{ji}^{(l)})^2$$

$(x^{(i)}, y^{(i)})$

Focusing on a single example  $x^{(i)}, y^{(i)}$ , the case of 1 output unit, and ignoring regularization ( $\lambda = 0$ ),

$$\text{cost}(i) = y^{(i)} \log h_\Theta(x^{(i)}) + (1 - y^{(i)}) \log h_\Theta(x^{(i)})$$

(Think of  $\text{cost}(i) \approx (h_\Theta(x^{(i)}) - y^{(i)})^2$ )

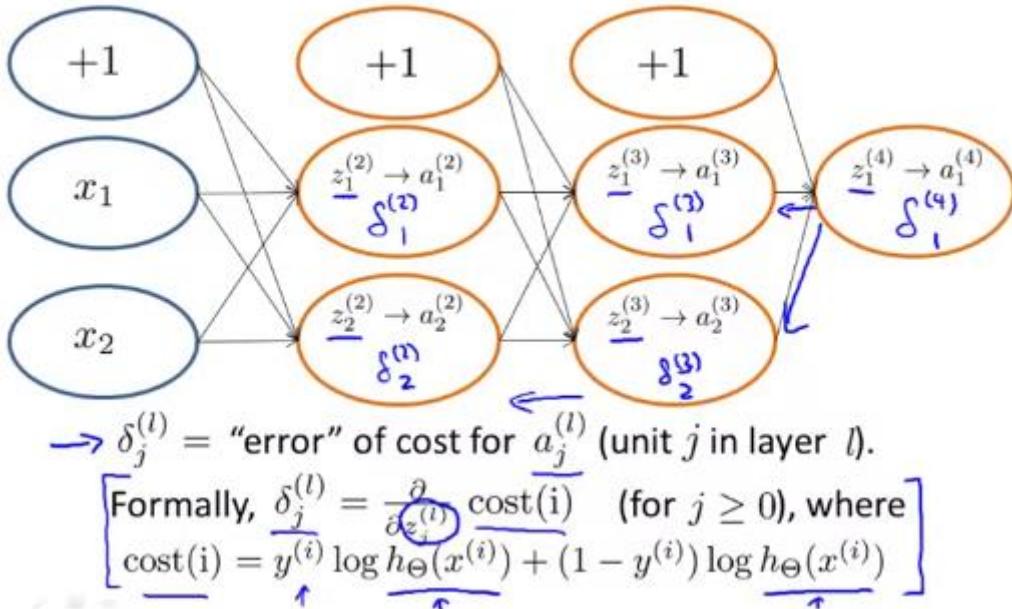
I.e. how well is the network doing on example i?

So **let's just focus on the single example  $x^{(i)}, y^{(i)}$**  and focus on the case of having one output unit, so  $y^{(i)}$  here is just a real number. And let's **ignore regularization**, so **lambda equals 0**. And this final term, that regularization term, goes away. Now if you look inside the summation, you find that the cost term associated with the  $i^{\text{th}}$  training example, that is the cost associated with the training example  $x^{(i)}, y^{(i)}$  that's going to be given by this expression. So, the cost of the training example  $i$  is written as follows. And what this cost function does is it plays a role similar to the squared error.

So, rather than looking at this complicated expression, if you want **you can think of cost of i** being approximately, you know, the **square difference between what the neural network output, versus what is the actual value**. Just as in logistic regression, we actually prefer to use the slightly more complicated cost function using the log. But for the purpose of intuition, feel free to think of the cost function as being that sort of **squared error cost function**. And so this **cost(i)** measures how well is the network doing on **correctly predicting example i**, how close is the output to the actual observed label  $\mathbf{y}^{(i)}$ .

Now let's look at **what back propagation is doing**.

### Forward Propagation



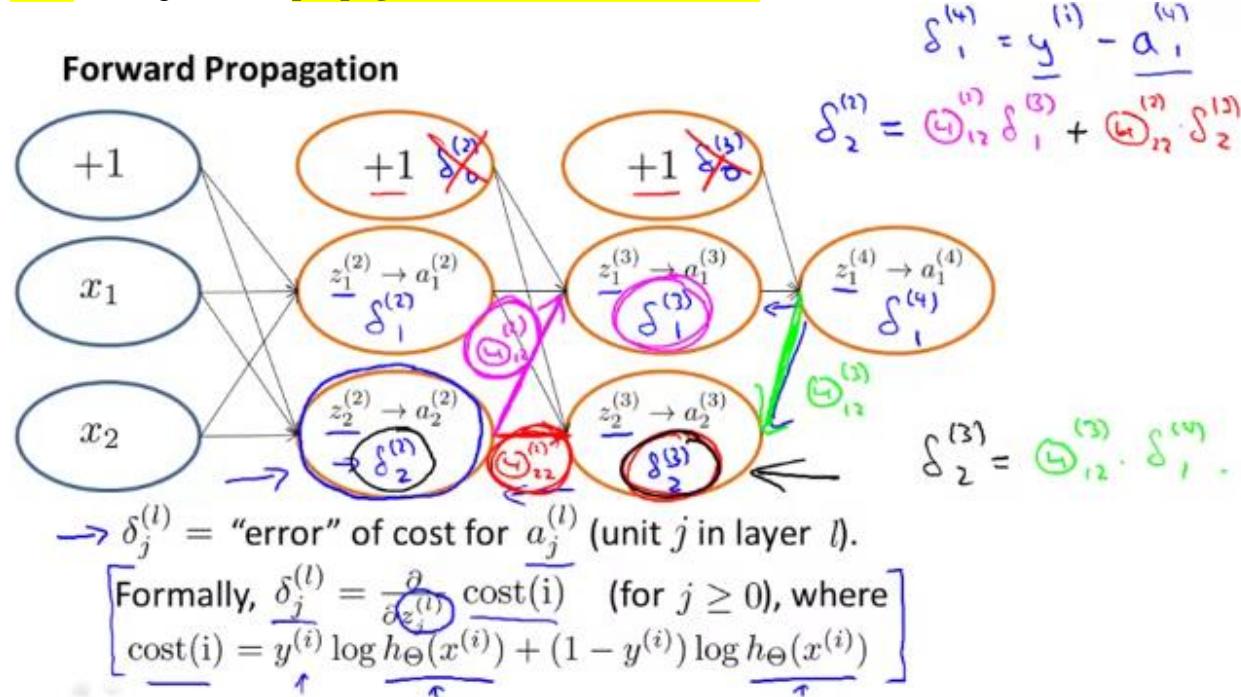
One useful intuition is that back propagation is computing these  $\delta_j^{(l)}$  terms. And we can **think of these as the "error" of the activation value that we got for unit j in the layer, in the l<sup>th</sup> layer**. More formally, for, and this is maybe only for those of you who are familiar with calculus, more formally, **what the delta terms actually are** is this, **they're the partial derivative with respect to  $z_j^{(l)}$** , that is this weighted sum of inputs that were computing these z terms, partial derivatives with respect to these things of the cost function. So concretely, the cost function is a function of the label y and of the value, this h(x), output value of neural network. And if we could go inside the neural network and just change those  $z_j^{(l)}$  values a little bit, then that will affect these values that the neural network is outputting. And so that will end up changing the cost function. And again really, this is only for those of you who are expert in calculus.

If you're familiar with, you're comfortable with partial derivatives, what these **delta terms** are is they turn out to be the **partial derivative of the cost function, with respect to these intermediate terms that were computing**. And so **they're a measure of how much would we like to change the neural network's weights, in order to affect these intermediate values of the computation, so as to affect the final output of the neural network h(x) and therefore affect the overall cost**.

In case this last part of this partial derivative intuition, in case that doesn't make sense, don't worry about it, the rest of this, we can do without really talking about partial derivatives. But let's look in more detail about what back propagation is doing.

For the output layer, it first sets this delta term,  $\delta^{(4)}_1$ , as  $y^{(i)}$  if we're doing forward propagation and back propagation on this training example  $i$ , that says  $y^{(i)}$  minus  $a^{(4)}_1$ . So this is really the error, right? It's the difference between the actual value of  $y$  minus what was the value predicted, and so we're gonna compute  $\delta^{(4)}_1$  like so.

Next we're gonna do, propagate these values backwards.



I'll explain this in a second, and end up computing the delta terms for the previous layer, we're gonna end up with  $\delta^{(3)}_1$   $\delta^{(3)}_2$ , and then we're gonna propagate this further backward, and end up computing  $\delta^{(2)}_1$   $\delta^{(2)}_2$ . Now the back propagation calculation is a lot like running the forward propagation algorithm, but doing it backwards.

So here's what I mean. Let's look at how we end up with this value of  $\delta^{(2)}_2$ . So we have  $\delta^{(2)}_2$ . And similar to forward propagation, let me label a couple of the weights. So this weight, which I'm going to draw in cyan, let's say that weight is  $\theta^{(2)}_{12}$ , and this one down here we highlight this in red, that is going to be let's say  $\theta^{(2)}_{22}$ . So if we look at how  $\delta^{(2)}_2$  is computed, how it's computed with this node, it turns out that what we're going to do, is gonna take this value and multiply it by this weight, and add it to this value multiplied by that weight. So it's really a weighted sum of these delta values weighted by the corresponding edge strength. So concretely, let me fill this in, this  $\delta^{(2)}_2$  is going to be equal to,  $\theta^{(2)}_{12}$  is that magenta weight times  $\delta^{(3)}_1$  plus, and the thing I had in red, that's  $\theta^{(2)}_{22}$  times  $\delta^{(3)}_2$ .

So it's really literally this red weight times this value, plus this magenta weight times this value, and that's how we wind up with that value of  $\delta$ . And just as another example, let's look at this value. How do we get that value? Well it's a similar process. If this weight, which I'm gonna highlight in green, if this weight is equal to, say,  $\theta^{(3)12}$ . Then we have that  $\delta^{(3)2}$  is going to be equal to that green weight,  $\theta^{(3)12}$  times  $\delta^{(4)1}$ . And by the way, so far I've been writing the delta values only for the hidden units, but excluding the bias units.

Depending on how you define the back propagation algorithm, or depending on how you implement it, you know, you may end up implementing something that computes delta values for these bias units as well. The bias units always output the value of plus one, and they are just what they are, and there's no way for us to change the value. And so, depending on your implementation of back prop, the way I usually implement it. I do end up computing these delta values, but we just discard them, we don't use them. Because they don't end up being part of the calculation needed to compute a derivative.

So hopefully that gives you a little better intuition about what back propagation is doing. In case all of this still seems sort of magical, sort of black box, in a later video, in the putting it together video, I'll try to get a little bit more intuition about what back propagation is doing. But unfortunately this is a difficult algorithm to try to visualize and understand what it is really doing. But fortunately you know, I've been, I guess many people have been using very successfully for many years. And if you implement the algorithm you can have a very effective learning algorithm. Even though the inner workings of exactly how it works can be harder to visualize.

## Summary – Back propagation Intuition

Note: Errors in video/transcript

- 4:39, the last term for the calculation for  $\mathbf{z}^{(3)1}$  (three-color handwritten formula) should be  $\mathbf{a}^{(2)2}$  instead of  $\mathbf{a}^{(2)1}$
- 6:08 - the equation for cost(i) is incorrect. The first term is missing parentheses for the log() function, and the second term should be  $(1-y^{(i)})\log(1-h_\theta(x^{(i)}))$
- 8:50 –  $\delta^{(4)} = y - a^{(4)}$  is incorrect and should be  $\delta^{(4)}=a^{(4)}-y$

Recall that the cost function for a neural network is:

$$J(\Theta) = -\frac{1}{m} \sum_{t=1}^m \sum_{k=1}^K \left[ y_k^{(t)} \log(h_\Theta(x^{(t)}))_k + (1 - y_k^{(t)}) \log(1 - h_\Theta(x^{(t)}))_k \right] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{j,i}^{(l)})^2$$

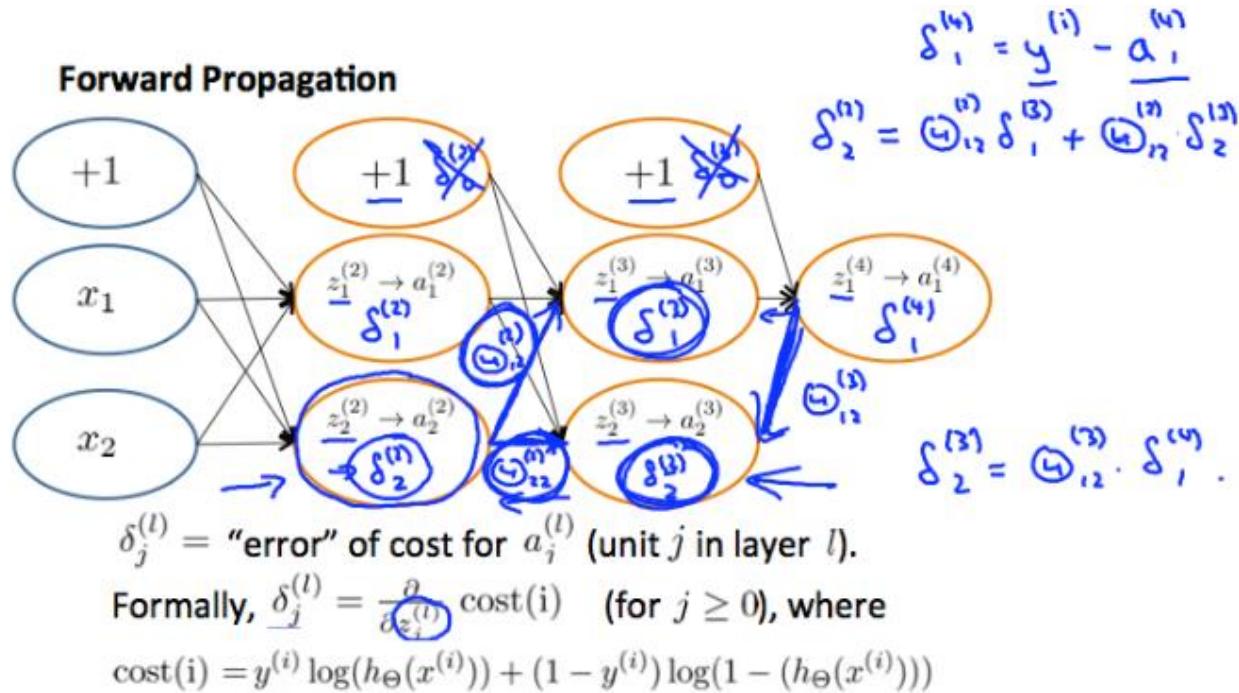
If we consider simple non-multiclass classification ( $k = 1$ ) and disregard regularization, the cost is computed with:

$$\text{cost}(t) = y^{(t)} \log(h_\Theta(x^{(t)})) + (1 - y^{(t)}) \log(1 - h_\Theta(x^{(t)}))$$

Intuitively,  $\delta^{(l)j}$  is the "error" for  $a^{(l)j}$  (unit j in layer l). More formally, the delta values are actually the derivative of the cost function:

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(t)$$

Recall that our derivative is the slope of a line tangent to the cost function, so the steeper the slope the more incorrect we are. Let us consider the following neural network below and see how we could calculate some  $\delta_j^{(l)}$



In the image above, to calculate  $\delta_2^{(2)}$ , we multiply the weights  $\Theta_{12}^{(2)}$  and  $\Theta_{22}^{(2)}$  by their respective  $\delta$  values found to the right of each edge. So we get  $\delta_2^{(2)} = \Theta_{12}^{(2)} * \delta_1^{(2)} + \Theta_{22}^{(2)} * \delta_2^{(2)}$ . To calculate every single possible  $\delta_j^{(l)}$ , we could **start from the right of our diagram**. We can think of our edges as our  $\Theta_{ij}$ . Going from right to left, to calculate the value of  $\delta_j^{(l)}$ , you can just take the overall sum of each weight times the  $\delta$  it is coming from. Hence, another example would be  $\delta_2^{(3)} = \Theta_{12}^{(3)} * \delta_1^{(3)}$ .

## Implementation Note: Unrolling Parameters

In the previous video, we talked about **how to use back propagation to compute the derivatives of your cost function**.

In this video, I want to quickly tell you about one implementational detail of **unrolling your parameters from matrices into vectors**, which we need in order to use the advanced optimization routines.

Concretely, let's say you've implemented a cost function that takes as input, you know, parameters  $\theta$  and returns the cost function and returns derivatives.

## Advanced optimization

```
function [jVal, gradient] = costFunction(theta)
    ...
optTheta = fminunc(@costFunction, initialTheta, options)
```

Neural Network ( $L=4$ ):

- $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$  - matrices (Theta1, Theta2, Theta3)
- $D^{(1)}, D^{(2)}, D^{(3)}$  - matrices (D1, D2, D3)

"Unroll" into vectors

Then you can pass this to an advanced optimization algorithm like **fminunc**, and fminunc isn't the only one by the way, there are also other advanced optimization algorithms. But what all of them do is take as input, pointedly cost function, and some initial value of  $\theta$ . And both, and these routines assume that  $\theta$  and the **initial value of  $\theta$** , that these **are parameter vectors**, maybe  $\mathbf{R}^n$  or  $\mathbf{R}^{n+1}$ , but these are vectors and it also assumes that, you know, your **cost function will return as a second return value this gradient** which is also  $\mathbf{R}^n$  and  $\mathbf{R}^{n+1}$ , so also a vector.

This worked fine when we were using logistic regression but now that we're using a neural network **our parameters are no longer vectors, but instead they are these matrices** where for a 4 layer neural network we would have parameter matrices  $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$  that we might represent in Octave as these **matrices  $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$** . And similarly these gradient terms that were expected to return, well, in the previous video we showed how to compute these **gradient matrices**, which was **capital D<sup>(1)</sup>, capital D<sup>(2)</sup>, capital D<sup>(3)</sup>**, which we might represent in Octave as **matrices D1, D2, D3**.

In this video I want to quickly tell you about the idea of **how to take these matrices and unroll them into vectors** so that they end up being in a format suitable for passing into as  $\theta$  here off, for getting out a gradient there.

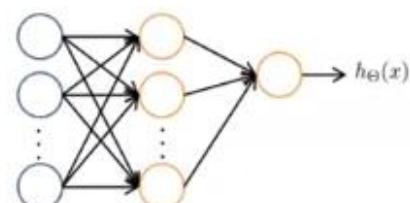
Concretely, let's say we have a neural network with **one input layer with ten units, hidden layer with ten units and one output layer with just one unit**.

### Example

$$s_1 = 10, s_2 = 10, s_3 = 1$$

$$\Theta^{(1)} \in \mathbb{R}^{10 \times 11}, \Theta^{(2)} \in \mathbb{R}^{10 \times 11}, \Theta^{(3)} \in \mathbb{R}^{1 \times 11}$$

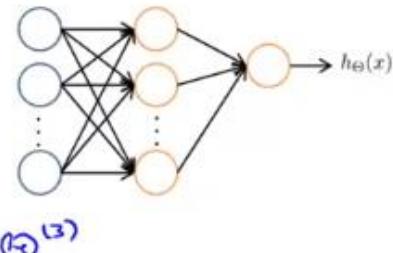
$$D^{(1)} \in \mathbb{R}^{10 \times 11}, D^{(2)} \in \mathbb{R}^{10 \times 11}, D^{(3)} \in \mathbb{R}^{1 \times 11}$$



So  $s_1$  is the number of units in layer one and  $s_2$  is the number of units in layer two, and  $s_3$  is a number of units in layer three. In this case, the dimension of your matrices  $\Theta$  and  $D$  are going to be given by these expressions.

For example,  $\Theta^{(1)}$  is going to be a 10 by 11 matrix and so on. So in Octave if you want to convert between these matrices and vectors, we can do is take your  $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ , and write this piece of code and this will take all the elements of your three  $\Theta$  matrices and take all the elements of  $\Theta^{(1)}$ , all the elements of  $\Theta^{(2)}$ , all the elements of  $\Theta^{(3)}$ , and unroll them and put all the elements into a big long vector, which is thetaVec and similarly the second command would take all of your  $D$  matrices and unroll them into a big long vector and call them DVec.

### Example

$$\begin{aligned} s_1 &= 10, s_2 = 10, s_3 = 1 \\ \rightarrow \Theta^{(1)} &\in \mathbb{R}^{10 \times 11}, \Theta^{(2)} \in \mathbb{R}^{10 \times 11}, \Theta^{(3)} \in \mathbb{R}^{1 \times 11} \\ \rightarrow D^{(1)} &\in \mathbb{R}^{10 \times 11}, D^{(2)} \in \mathbb{R}^{10 \times 11}, D^{(3)} \in \mathbb{R}^{1 \times 11} \\ \rightarrow \text{thetaVec} &= [\text{Theta1}(:); \text{Theta2}(:); \text{Theta3}(:)]; \\ \rightarrow \text{DVec} &= [D1(:); D2(:); D3(:)]; \\ \text{Theta1} &= \text{reshape}(\text{thetaVec}(1:110), 10, 11); \\ \rightarrow \text{Theta2} &= \text{reshape}(\text{thetaVec}(111:220), 10, 11); \\ \rightarrow \text{Theta3} &= \text{reshape}(\text{thetaVec}(221:231), 1, 11); \end{aligned}$$


And finally if you want to go back from the vector representations to the matrix representations, what you do to get back to  $\Theta^{(1)}$  say is take thetaVec and pull out the first 110 elements. So  $\Theta^{(1)}$  has 110 elements because it's a 10 by 11 matrix so that pulls out the first 110 elements and then you can use the reshape command to reshape those back into  $\Theta^{(1)}$ . And similarly, to get back  $\Theta^{(2)}$  you pull out the next 110 elements and reshape it. And for  $\Theta^{(3)}$ , you pull out the final 11 elements and run reshape to get back  $\Theta^{(3)}$ .

Here's a quick Octave demo of that process.

So for this example let's set Theta1 equal to be ones(10, 11), so it's a matrix of all ones.

```

Octave-3.2.4
For more information, visit http://www.octave.org/help-wanted.html
Report bugs to <bug@octave.org> (but first, please read
http://www.octave.org/bugs.html to learn how to write a helpful report).
For information about changes from previous versions, type 'news'.
octave-3.2.4.exe:1> PS1('>> ')
>> Theta1 = ones(10,11)
Theta1 =

```

1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1

```

>> T

```

And just to make that easier to see, let's set **Theta2** to be **2 \* ones(10, 11)**.

```

>> Theta2 = 2*ones(10,11)
Theta2 =

```

2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2

And let's set **Theta3** equals **3 \* ones(1, 11)**.

```

>> Theta3 = 3*ones(1,11)
Theta3 =

```

3	3	3	3	3	3	3	3	3	3	3
---	---	---	---	---	---	---	---	---	---	---

```

>> the

```

So this is **3 separate matrices**: **Theta1**, **Theta2**, **Theta3**.

If you want to put all of these into a vector, **ThetaVec** equals **Theta1(:,)**, **Theta2(:,)**, **Theta3(:)**.

Right, that's a colon in the middle and like so, and now **ThetaVec** is going to be a very long vector. That's 231 elements.

```
>> thetaVec = [ Theta1(:); Theta2(:); Theta3(:) ];
>> size(thetaVec)
ans =
    231      1
```

If I display it, I find that this **very long vector** with **all the elements** of the first matrix, all the elements of the second matrix, then all the elements of the third matrix.

The screenshot shows a MATLAB command window titled 'Len'. It displays the command `>> thetaVec` at the top. The output is a long vector of ones, starting with `1` and ending with `2`. Below the output, the status bar shows 'Lines 111-132' and '4:47 / 7:47'. The window has a standard Windows-style title bar and taskbar.

And if I want to get back my original matrices, I can do **reshape thetaVec**. Let's pull out the first 110 elements and reshape them to a 10 by 11 matrix. This gives me back theta 1.

```
>> reshape(thetaVec(1:110), 10,11)
ans =
    1   1   1   1   1   1   1   1   1   1
    1   1   1   1   1   1   1   1   1   1
    1   1   1   1   1   1   1   1   1   1
    1   1   1   1   1   1   1   1   1   1
    1   1   1   1   1   1   1   1   1   1
    1   1   1   1   1   1   1   1   1   1
    1   1   1   1   1   1   1   1   1   1
    1   1   1   1   1   1   1   1   1   1
    1   1   1   1   1   1   1   1   1   1
    1   1   1   1   1   1   1   1   1   1
```

And if I then pull out the next 110 elements. So that's indices 111 to 220. I get back all of my 2's.

```
>> reshape(thetaVec(111:220), 10,11)
ans =
    2   2   2   2   2   2   2   2   2   2
    2   2   2   2   2   2   2   2   2   2
    2   2   2   2   2   2   2   2   2   2
    2   2   2   2   2   2   2   2   2   2
    2   2   2   2   2   2   2   2   2   2
    2   2   2   2   2   2   2   2   2   2
    2   2   2   2   2   2   2   2   2   2
    2   2   2   2   2   2   2   2   2   2
    2   2   2   2   2   2   2   2   2   2
```

And if I go from 221 up to the last element, which is element 231, and reshape to 1 by 11, I get back theta 3.

```
>> reshape(thetaVec(221:231), 1,11)
ans =
    3    3    3    3    3    3    3    3    3    3    3
>>
```

To make this process really concrete, here's how we use the unrolling idea to implement our learning algorithm. Let's say that you have some initial value of the parameters  $\theta_1, \theta_2, \theta_3$ . What we're going to do is take these and unroll them into a long vector we're gonna call `initialTheta` to pass in to `fminunc` as this initial setting of the parameters theta.

### Learning Algorithm

- Have initial parameters  $\underline{\theta^{(1)}, \theta^{(2)}, \theta^{(3)}}$ .
- Unroll to get `initialTheta` to pass to
- `fminunc (@costFunction, initialTheta, options)`

The other thing we need to do is implement the cost function. Here's my implementation of the cost function.

```
function [jval, gradientVec] = costFunction(thetaVec)
    From thetaVec, get  $\theta^{(1)}, \theta^{(2)}, \theta^{(3)}$ .
    Use forward prop/back prop to compute  $D^{(1)}, D^{(2)}, D^{(3)}$  and  $J(\theta)$ .
    Unroll  $D^{(1)}, D^{(2)}, D^{(3)}$  to get gradientVec.
```

The cost function is going to give us, input `thetaVec`, which is going to be all of my parameters vectors that in the form that's been unrolled into a vector. So the first thing I'm going to do is I'm going to use `thetaVec` and I'm going to use the `reshape` functions, so I'll pull out elements from `thetaVec` and use `reshape` to get back my original parameter matrices,  $\theta^{(1)}, \theta^{(2)}, \theta^{(3)}$ . So these are going to be matrices that I'm going to get. So that gives me a more convenient form in which to use these matrices so that I can run forward propagation and back propagation to compute my derivatives, and to compute my cost function  $J(\theta)$ .

And finally, I can then take my derivatives and unroll them, to keeping the elements in the same ordering as I did when I unroll my thetas. But I'm gonna unroll  $D1, D2, D3$ , to get `gradientVec` which is now what my cost function can return. It can return a vector of these derivatives.

So, hopefully, you now have a good sense of how to convert back and forth between the matrix representations of the parameters versus the vector representation of the parameters. The advantage of the matrix representation is that when your parameters are stored as matrices it's more convenient when you're doing forward propagation and back propagation and it's easier when your parameters are stored as matrices to take advantage of the, sort of, vectorized implementations. Whereas in contrast the advantage of the vector representation, when you have like `thetaVec` or `DVec` is that when you are using the advanced optimization algorithms, those algorithms tend to assume that you have all of your parameters unrolled into a big long vector.

And so with what we just went through, hopefully you can now quickly convert between the two as needed.

## Summary - Implementation Note: Unrolling Parameters

With neural networks, we are working with **sets of matrices**:

$$\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}, \dots$$

$$D^{(1)}, D^{(2)}, D^{(3)}, \dots$$

In order to use optimizing functions such as "**fminunc()**", we will want to "**unroll**" all the elements and put them **into one long vector**:

```
1 thetaVector = [ Theta1(:); Theta2(:); Theta3(:); ]
2 deltaVector = [ D1(:); D2(:); D3(:) ]
```

If the dimensions of **Theta1** is **10x11**, **Theta2** is **10x11** and **Theta3** is **1x11**, then we can **get back our original matrices** from the "unrolled" versions as follows:

```
1 Theta1 = reshape(thetaVector(1:110),10,11)
2 Theta2 = reshape(thetaVector(111:220),10,11)
3 Theta3 = reshape(thetaVector(221:231),1,11)
```

To summarize:

### Learning Algorithm

- Have initial parameters  $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ .
- Unroll to get **initialTheta** to pass to
- **fminunc(@costFunction, initialTheta, options)**
- function [**jval, gradientVec**] = costFunction(**thetaVec**)
 From **thetaVec**, get  $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ .
 Use forward prop/back prop to compute  $D^{(1)}, D^{(2)}, D^{(3)}$  and  $J(\Theta)$ .
 Unroll  $D^{(1)}, D^{(2)}, D^{(3)}$  to get **gradientVec**.

## Gradient Checking

In the last few videos we talked about how to do forward propagation and back propagation in a neural network in order to compute derivatives.

But back prop as an algorithm has a lot of details and, you know, can be a little bit tricky to implement. And one unfortunate property is that there are many ways to have subtle bugs in back prop. So that if you run it with gradient descent or some other optimizational algorithm, it could actually look like it's working, and your cost function,  $J(\theta)$  may end up decreasing on every iteration of gradient descent. But this could prove true even though there might be some bug in your implementation of back prop. So that it looks  $J(\theta)$  is decreasing, but you might just wind up with a neural network that has a higher level of error than you would with a bug free implementation. And you might just not know that there was this subtle bug that was giving you worse performance.

So, what can we do about this? There's an idea called **gradient checking** that eliminates almost all of these problems.

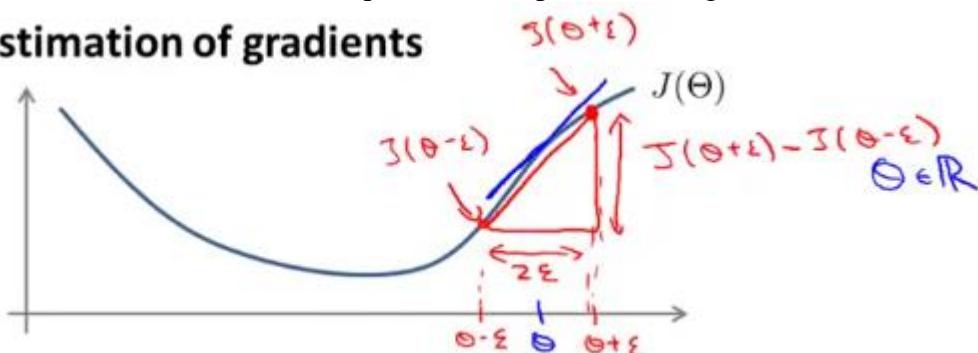
So, today every time I implement back propagation or a similar gradient descent algorithm on a neural network or any other reasonably complex model, I always implement gradient checking.

And if you do this, it will help you make sure and sort of gain high confidence that your implementation of forward prop and back prop or whatever is 100% correct. And from what I've seen this pretty much eliminates all the problems associated with a sort of a buggy implementation as a back prop. And in the previous videos I asked you to take on faith that the formulas I gave for computing the deltas and the vs and so on, I asked you to take on faith that those actually do compute the gradients of the cost function. But once you implement numerical gradient checking, which is the topic of this video, you'll be able to absolutely verify for yourself that the code you're writing does indeed, is indeed computing the derivative of the cross function  $J$ .

**So here's the idea,** consider the following example.

Suppose that I have the function  $J(\theta)$  and I have some value  $\theta$  and for this example gonna **assume** that  $\theta$  is just a real number. And let's say that I want to estimate the derivative of this function at this point, and so the derivative is equal to the slope of that tangent line.

## Numerical estimation of gradients



Here's how I'm going to numerically approximate the derivative, or rather here's a procedure for numerically approximating the derivative. I'm going to compute  $\theta + \epsilon$ , so now we move it to the right, and I'm gonna compute  $\theta - \epsilon$ , and I'm going to look at those two points, and connect them by a straight line, and I'm gonna connect these two points by a straight line, and I'm gonna use the **slope of that little red line** as my **approximation to the derivative**, which is the **true derivative** is the slope of that blue line over there. So, you know it seems like it would be a pretty good approximation. Mathematically, the slope of this red line is this vertical height divided by this horizontal width. So this point on top is the  $J(\theta + \epsilon)$ . This point here is  $J(\theta - \epsilon)$ , so this vertical difference is  $J(\theta + \epsilon)$  minus  $J(\theta - \epsilon)$  and this horizontal distance is just  $2\epsilon$ . So my approximation is going to be that the derivative with respect of  $\theta$  of  $J(\theta)$  at this value of  $\theta$ , that that's approximately  $(J(\theta + \epsilon) - J(\theta - \epsilon)) / 2\epsilon$ .

$$\frac{\partial}{\partial \theta} J(\theta) \approx \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon}$$

Usually, I use a pretty **small value for  $\epsilon$** , expect epsilon to be maybe on the order of 10 to the minus 4.

$$\epsilon = 10^{-4}$$

There's usually a large range of different values for epsilon that work just fine. And in fact, **if you let epsilon become really small**, then mathematically **this term here, actually mathematically, it becomes the derivative. It becomes exactly the slope of the function at**

this point. It's just that we don't want to use epsilon that's too, too small, because then you might run into numerical problems. So I usually use epsilon around ten to the minus four.

And by the way some of you may have seen an alternative formula for estimating the derivative which is this formula. This one on the right is called a one-sided difference, whereas the formula on the left, that's called a two-sided difference.

$$\frac{\partial}{\partial \theta} J(\theta) \approx \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon} \quad \leftarrow \quad \cancel{\frac{J(\theta + \epsilon) - J(\theta)}{\epsilon}}$$

$\epsilon = 10^{-4} \leftarrow$

The two sided difference gives us a slightly more accurate estimate, so I usually use that, rather than this one sided difference estimate.

So, concretely, when you implement in Octave, if you implement the following, you implement code to compute gradApprox, which is going to be our approximation to the derivative as just here this formula,  $(J(\theta + \epsilon) - J(\theta - \epsilon)) / 2\epsilon$ . And this will give you a numerical estimate of the gradient at that point. And in this example it seems like it's a pretty good estimate.

Now on the previous slide, we considered the case of when  $\theta$  was a rolled number. Now let's look at a more general case of when  $\theta$  is a vector parameter, so let's say  $\theta$  is an  $\mathbb{R}^n$ , and it might be an unrolled version of the parameters of our neural network. So  $\theta$  is a vector that has  $n$  elements,  $\theta_1$  up to  $\theta_n$ .

### Parameter vector $\theta$

$$\begin{aligned} & \rightarrow \theta \in \mathbb{R}^n \quad (\text{E.g. } \theta \text{ is "unrolled" version of } \underline{\theta^{(1)}}, \underline{\theta^{(2)}}, \underline{\theta^{(3)}}) \\ & \rightarrow \theta = [\theta_1, \theta_2, \theta_3, \dots, \theta_n] \\ & \rightarrow \frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J(\theta_1 + \epsilon, \theta_2, \theta_3, \dots, \theta_n) - J(\theta_1 - \epsilon, \theta_2, \theta_3, \dots, \theta_n)}{2\epsilon} \\ & \rightarrow \frac{\partial}{\partial \theta_2} J(\theta) \approx \frac{J(\theta_1, \theta_2 + \epsilon, \theta_3, \dots, \theta_n) - J(\theta_1, \theta_2 - \epsilon, \theta_3, \dots, \theta_n)}{2\epsilon} \\ & \quad \vdots \\ & \rightarrow \frac{\partial}{\partial \theta_n} J(\theta) \approx \frac{J(\theta_1, \theta_2, \theta_3, \dots, \theta_n + \epsilon) - J(\theta_1, \theta_2, \theta_3, \dots, \theta_n - \epsilon)}{2\epsilon} \end{aligned}$$

We can then use a similar idea to approximate all the partial derivative terms. Concretely the partial derivative of a cost function with respect to the first parameter,  $\theta_1$ , that can be obtained by

taking  $J$  and increasing  $\theta_1$ . So you have  $J(\theta_1 + \epsilon \text{ and so on}) \text{ minus } J(\theta_1 - \epsilon \text{ and so on}) \text{ divided by } 2\epsilon$ . The partial derivative respect to the second parameter  $\theta_2$  is again this thing except that you would take  $J$  of here you're increasing  $\theta_2$  by  $\epsilon$ , and here you're decreasing  $\theta_2$  by  $\epsilon$  and so on down to the derivative with respect of  $\theta_n$  would give you increase and decrease  $\theta_n$  by  $\epsilon$  over there. So, these equations give you a way to numerically approximate the partial derivative of  $J$  with respect to any one of your parameters  $\theta_i$ .

Concretely, what you implement is therefore the following. We implement the following in octave to numerically compute the derivatives.

```
for i = 1:n, 
    thetaPlus = theta;
    thetaPlus(i) = thetaPlus(i) + EPSILON;
    thetaMinus = theta;
    thetaMinus(i) = thetaMinus(i) - EPSILON;
    gradApprox(i) = (J(thetaPlus) - J(thetaMinus))
                    / (2*EPSILON);
end;
```

Check that  $\text{gradApprox} \approx \text{DVec}$

We say, for  $i = 1:n$ , where  $n$  is the dimension of our parameter vector  $\theta$ , and I usually do this with the unrolled version of the parameter. So  $\theta$  is just a long list of all of my parameters in my neural network, say.

I'm gonna set  $\text{thetaPlus} = \theta$ , then increase  $\text{thetaPlus}$  of the  $i^{\text{th}}$  element by  $\epsilon$ , and so this is basically  $\text{thetaPlus}$  is equal to  $\theta$  except for  $\text{thetaPlus}(i)$  which is now incremented by  $\epsilon$ . So  $\text{thetaPlus}$  is equal to,  $\theta_1, \theta_2$  and so on, then  $\theta_i$  has  $\epsilon$  added to it and then we go down to  $\theta_n$ . So this is what  $\text{thetaPlus}$  is.

And similarly, these two lines set  $\text{thetaMinus}$  to something similar except that this instead of  $\theta_i + \epsilon$ , this now becomes  $\theta_i - \epsilon$ .

And then finally, you implement this  $\text{gradApprox}(i)$  and this would give you your approximation to the partial derivative with respect to  $\theta_i$  of  $J(\theta)$ . And the way we use this in our neural network implementation is we would implement this, implement this for loop to compute the partial derivative of the cost function with respect to every parameter in that network, and we can then take the gradient that we got from back prop, so  $\text{DVec}$  was the derivative we got from back prop. All right, so back prop, back propagation, was a relatively efficient way to compute the derivatives or partial derivatives of a cost function with respect to all our parameters.

And what I usually do is then, take my **numerically computed derivative** that is this **gradApprox** that we just have from up here, and make sure that that is equal or approximately equal up to small values of numerical round off, that it's pretty close to the **DVec** that I got from back prop. And if these two ways of computing the derivative give me the same answer, or give me very similar answers up to a few decimal places, then I'm much more confident that my implementation of back prop is correct. And when I plug these DVec vectors into gradient descent or some advanced optimization algorithm, I can then be much more confident that I'm computing the derivatives correctly, and therefore that hopefully my code will run correctly and do a good job optimizing  $J(\theta)$ .

Finally, I wanna put everything together and tell you how to implement this numerical gradient checking. Here's what I usually do.

### **Implementation Note:**

- Implement backprop to compute **DVec** (unrolled  $D^{(1)}, D^{(2)}, D^{(3)}$ ).
- Implement numerical gradient check to compute **gradApprox**.
- Make sure they give similar values.
- Turn off gradient checking. Using backprop code for learning.

First thing I do is **implement back propagation** to **compute DVec**. So there's a procedure we talked about in the earlier video to compute DVec which may be our unrolled version of these matrices. Then what I do, is **implement a numerical gradient checking** to **compute gradApprox**. So this is what I described earlier in this video and in the previous slide. Then should **make sure that DVec and gradApprox give similar values**, you know let's say up to a few decimal places. And finally and this is the important step, before you start to use your code for learning, for seriously training your network, it's important to **turn off gradient checking** and **to no longer compute this gradApprox** thing using the numerical derivative formulas that we talked about earlier in this video.

And the reason for that is the numeric code, gradient checking code, the stuff we talked about in this video, that's a very **computationally expensive**, that's a very slow way to try to approximate the derivative. Whereas in contrast, the back propagation algorithm that we talked about earlier, that is the thing we talked about earlier for computing,  $D1, D2, D3$ , for Dvec, **back prop is a much more computationally efficient way of computing for derivatives**. So once you've verified that your implementation of back propagation is correct, you should turn off gradient checking and just stop using that.

### **Important:**

- - Be sure to disable your gradient checking code before training your classifier. If you run numerical gradient computation on every iteration of gradient descent (or in the inner loop of **costFunction(...)**) your code will be very slow.

So just to reiterate, **you should be sure to disable your gradient checking code before running your algorithm for many iterations of gradient descent** or for many iterations of the advanced optimization algorithms, in order to train your classifier.

Concretely, if you were to run the numerical gradient checking on every single iteration of gradient descent, or if you were in the inner loop of your costFunction, then your code would be very slow. Because the numerical gradient checking code is much slower than the back propagation algorithm, than the back propagation method where, you remember, we were computing  $\delta^{(4)}, \delta^{(3)}, \delta^{(2)}$ , and so on, that was the back propagation algorithm, that is a much faster way to compute derivatives than gradient checking. So when you're ready, once you've verified the implementation of back propagation is correct, make sure you turn off or you disable your gradient checking code while you train your algorithm, or else your code could run very slowly.

So, that's how you take gradients numerically, and that's how you can verify that implementation of back propagation is correct. Whenever I implement back propagation or similar gradient descent algorithm for a complicated model I always do this gradient checking and this really helps me make sure that my code is correct.

## Summary - Gradient Checking

Gradient checking will assure that our back propagation works as intended. We can approximate the derivative of our cost function with:

$$\frac{\partial}{\partial \theta} J(\theta) \approx \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon}$$

With multiple theta matrices, we can approximate the derivative **with respect to  $\Theta_j$**  as follows:

$$\frac{\partial}{\partial \theta_j} J(\theta) \approx \frac{J(\theta_1, \dots, \theta_j + \epsilon, \dots, \theta_n) - J(\theta_1, \dots, \theta_j - \epsilon, \dots, \theta_n)}{2\epsilon}$$

A small value for  $\epsilon$  (epsilon) such as  $\epsilon = 10^{-4}$ , guarantees that the math works out properly. If the value for  $\epsilon$  is too small, we can end up with numerical problems.

Hence, we are only adding or subtracting epsilon to the  $\Theta_j$  matrix. In octave we can do it as follows:

```

1  epsilon = 1e-4;
2  for i = 1:n,
3    thetaPlus = theta;
4    thetaPlus(i) += epsilon;
5    thetaMinus = theta;
6    thetaMinus(i) -= epsilon;
7    gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2*epsilon)
8  end;
9

```

We previously saw how to calculate the deltaVector. So once we compute our gradApprox vector, we can check that **gradApprox ≈ deltaVector**.

Once you have verified **once** that your backpropagation algorithm is correct, you don't need to compute gradApprox again. The code to compute gradApprox can be very slow.

## Random Initialization

In the previous video, we've put together almost all the pieces you need in order to implement and train neural network.

There's just one last idea I need to share with you, which is the idea of **random initialization**.

When you're running an algorithm of gradient descent, or also the advanced optimization algorithms, we need to **pick some initial value for the parameters theta**. So for the advanced optimization algorithm, it assumes you will pass it some initial value for the parameters theta.

### Initial value of $\Theta$

For gradient descent and advanced optimization method, need initial value for  $\Theta$ .

```
optTheta = fminunc(@costFunction,
    initialTheta, options)
```

Now let's consider a gradient descent. For that, we'll also need to initialize theta to something, and then we can slowly take steps to go downhill using gradient descent, to go downhill, to minimize the function  $J(\Theta)$ ..

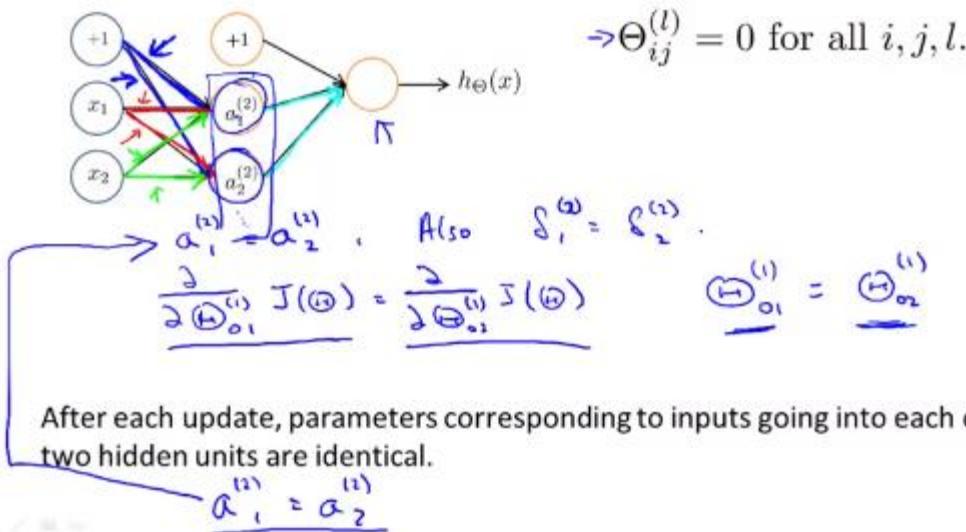
### Consider gradient descent

Set `initialTheta = zeros(n,1)` ?

So what can we set the initial value of theta to? Is it possible to set the initial value of theta to the vector of all zeros? Whereas this worked okay when we were using logistic regression, **initializing all of your parameters to zero actually does not work** when you are training a neural network.

Consider training the follow neural network, and let's say we initialize all the parameters of the network to 0.

## Zero initialization



And if you do that, then what you, what that means is that at the initialization, this **blue weight**, colored in blue is gonna equal to that weight, so they're **both 0**. And this weight that I'm coloring in in **red**, is equal to that weight, colored in red, and also this weight, which I'm coloring in **green** is going to equal to the value of that weight. And what that means is that both of your hidden units,  $a_1^{(2)}$  and  $a_2^{(2)}$ , are going to be computing the same function of your inputs. And thus you end up with, for every one of your training examples, you end up with  $a_1^{(2)}$  equals  $a_2^{(2)}$ . And moreover because I'm not going to show this in too much detail, but because these outgoing weights are the same you can also show that the delta values are also going to be the same. So concretely you end up with  $\delta_1^{(2)}$  equals  $\delta_2^{(2)}$ .

And if you work through the map further, what you can show is that the partial derivatives with respect to your parameters will satisfy the following that the partial derivative of the cost function with respect to, I'm breaking out the derivatives with respect to these two blue weights in your network, you find that these two partial derivatives are going to be equal to each other. And so what this means is that even after say one gradient descent update, you're going to update, say, this first blue weight was learning rate times this, and you're gonna update the second blue weight with some learning rate times this. And what this means is that even after one gradient descent update, those two blue weights, those two blue color parameters will end up the same as each other. So there'll be some nonzero value now, but this value would equal to that value.

And similarly, even after one gradient descent update, this value would equal to that value. There'll still be some non-zero values, just that the two red values are equal to each other. And similarly, the two green weights, well, they'll both change values, but they'll both end up with the same value as each other. So after each update, the parameters corresponding to the inputs going into each of the two hidden units are identical. That's just saying that the two green weights are still the same, the two red weights are still the same, the two blue weights are still the same, and what that means is that even after one iteration of, say gradient descent, you find that your two headed units are still computing exactly the same functions of the inputs. You still have this  $a_1^{(2)} = a_2^{(2)}$ , and so you're back to this case.

And as you keep running gradient descent, the blue weights,, the two blue weights, will stay the same as each other, the two red weights will stay the same as each other and the two green weights will stay the same as each other. And what this means is that your neural network really can compute very interesting functions, right?

Imagine that you had not only two hidden units, but imagine that you had many, many hidden units. Then what this is saying is that all of your hidden units are computing the exact same feature. All of your hidden units are computing the exact same function of the input.

And this is a highly redundant representation because that means that your final logistic regression unit, you really don't guess to see only one feature, because all of these are the same. And this prevents your neural network from learning something interesting.

In order to get around this problem, the way we initialize the parameters of a neural network therefore is with random initialization.

Concretely, the problem was saw on the previous slide is something called the **problem of symmetric weights**, that's the weights are being the same. So this random initialization is how we perform **symmetry breaking**. So what we do is we **initialize each value of  $\theta$**  to a random number between minus  $\epsilon$  and  $\epsilon$ . So this is a notation between numbers, between minus  $\epsilon$  and plus  $\epsilon$ . So my weight for my **parameters** are all going to be **randomly initialized between** minus  $\epsilon$  and plus  $\epsilon$ . The way I write code to do this in octave is I've said **Theta1** should be equal to this. So this **rand(10, 11)**, that's how you compute a random 10 by 11 dimensional matrix. All the values are between 0 and 1, so these are going to be raw numbers that take on any continuous values between 0 and 1. And so if you take a number between zero and one, multiply it by two times INIT\_EPSILON then minus INIT\_EPSILON, then you end up with a number that's between minus  $\epsilon$  and plus  $\epsilon$ . And this  $\epsilon$  here has nothing to do with the  $\epsilon$  that we were using when we were doing gradient checking. So when numerical gradient checking, there we were adding some values of  $\epsilon$  and  $\theta$ . This is your unrelated value of  $\epsilon$ . We just wanted to notate INIT\_EPSILON just to distinguish it from the value of  $\epsilon$  we were using in gradient checking.

And similarly if you want to initialize **Theta2** to a random 1 by 11 matrix you can do so using this piece of code here.

**So to summarize**, to create a neural network what you should do is randomly initialize the weights to small values close to zero, between -  $\epsilon$  and +  $\epsilon$  say. And then implement back propagation, do gradient checking, and use either gradient descent or one of the advanced optimization algorithms to try to minimize  $J(\theta)$  as a function of the parameters  $\theta$  starting from just randomly chosen initial value for the parameters. And by doing symmetry breaking, which is this process, hopefully gradient descent or the advanced optimization algorithms will be able to **find a good value of  $\theta$** .

## Summary - Random Initialization

Initializing all theta weights to zero does not work with neural networks. When we back propagate, all nodes will update to the same value repeatedly. Instead we can randomly initialize our weights for our  $\Theta$  matrices using the following method:

### Random initialization: Symmetry breaking

- Initialize each  $\Theta_{ij}^{(l)}$  to a random value in  $[-\epsilon, \epsilon]$   
(i.e.  $-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$ )
- E.g.  $\text{rand}(10,11) * (2 * \text{INIT\_EPSILON}) - \text{INIT\_EPSILON}$ ; Random  $10 \times 11$  matrix (betw. 0 and 1)
- $\text{Theta1} = \text{rand}(10,11) * (2 * \text{INIT\_EPSILON}) - \text{INIT\_EPSILON}; [-\epsilon, \epsilon]$
- $\text{Theta2} = \text{rand}(1,11) * (2 * \text{INIT\_EPSILON}) - \text{INIT\_EPSILON};$

Hence, we initialize each  $\Theta_{ij}^{(l)}$  to a random value between  $[-\epsilon, \epsilon]$ . Using the above formula guarantees that we get the desired bound. The same procedure applies to all the  $\Theta$ 's. Below is some working code you could use to experiment.

```

1 If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is
   1x11.
2
3 Theta1 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
4 Theta2 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
5 Theta3 = rand(1,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
6

```

**rand(x,y)** is just a function in octave that will initialize a matrix of random real numbers between 0 and 1.

(Note: the epsilon used above is unrelated to the epsilon from Gradient Checking)

## Putting It Together

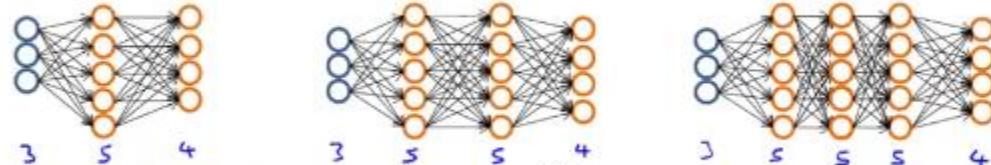
So, it's taken us a lot of videos to get through the neural network learning algorithm.

In this video, what I'd like to do is try to put all the pieces together, to give a overall summary or a bigger picture view, of how all the pieces fit together and of the overall process of how to implement a neural network learning algorithm.

When training a neural network, the **first thing** you need to do is pick some network architecture and by **architecture** I just mean **connectivity pattern between the neurons**.

### Training a neural network

Pick a network architecture (connectivity pattern between neurons)



No. output units: Number of classes

So, you know, we might choose between say, a neural network with 3 input units and 5 hidden units and 4 output units, versus one of 3, 5 hidden, 5 hidden, 4 outputs, and here are 3, 5, 5, 5 units in each of three hidden layers and 4 output units, and so these choices of how many hidden units in each layer and how many hidden layers, those are architecture choices.

So, how do you make these choices?

Well first, the number of input units, well that's pretty well defined. And once you decided on the fix set of **features  $x$**  the **number of input units** will just be, you know, the **dimension of your features  $x^{(i)}$** , would be determined by that. And if you are doing **multiclass classifications** the number of output units will be determined by the **number of classes in your classification problem**.

And just a reminder if you have a multiclass classification where  $y$  takes on say values between 1 and 10, so that you have ten possible classes. Then remember to rewrite your outputs  $y$  as these were the vectors. So instead of class one, you recode it as a vector like that, or for the second class you recode it as a vector like that.

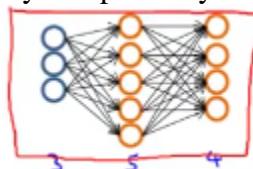
$$y \in \{1, 2, 3, \dots, 10\}$$

~~$y = 5$~~

$$y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \leftarrow \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 0 \end{bmatrix}$$

So **if one of the examples takes on the fifth class**, you know,  **$y$  equals 5**, then what you're showing to your neural network is **not actually a value of  $y$  equals 5**, instead here at the upper layer which would have ten output units, you will instead feed to the vector which you know with **one in the fifth position** and a bunch of zeros down here. So the choice of **number of input units** and **number of output units** is maybe somewhat **reasonably straightforward**.

And as for the number of hidden units and the number of hidden layers, a reasonable default is to use a **single hidden layer** and so this type of neural network shown on the left with just one hidden layer is probably the most common.



**Reasonable default: 1 hidden layer, or if >1 hidden layer, have same no. of hidden units in every layer (usually the more the better)**

Or if you use more than one hidden layer, again the reasonable default will be to have the same number of hidden units in every single layer. So here we have two hidden layers and each of these hidden layers have the same number five of hidden units and here we have, you know, three hidden layers and each of them has the same number that is five hidden units. Rather than doing this sort of network architecture on the left would be a perfect ably reasonable default. And as for the number of hidden units - usually, **the more hidden units the better**; it's just that if you have a lot of hidden units, it can become more computationally expensive, but very often, having more hidden units is a good thing.

And usually the **number of hidden units in each layer** will be maybe **comparable to the dimension of  $x$** , comparable to the number of features, or it could be any where from same number of hidden units as of input features to maybe twice of that or three or four times of that. So having the number of hidden units is comparable. You know, several times, or some what bigger than the number of input features is often a useful thing to do.

So, hopefully this gives you one reasonable set of default choices for network architecture and if you follow these guidelines, you will probably get something that works well, but in a later set of videos where I will talk specifically about advice for how to apply learning algorithms, I will actually say a lot more about how to choose a neural network architecture. Or actually have quite a lot I want to say later about how to make good choices for the number of hidden units, the number of hidden layers, and so on.

Next, here's what we need to implement in order to train a neural network, there are actually **six steps** that I have; I have **four on this slide** and **two more steps on the next slide**.

### Training a neural network

1. Randomly initialize weights
  2. Implement forward propagation to get  $h_{\Theta}(x^{(i)})$  for any  $x^{(i)}$
  3. Implement code to compute cost function  $J(\Theta)$
  4. Implement backprop to compute partial derivatives  $\frac{\partial}{\partial \Theta_{jk}^{(i)}} J(\Theta)$
- First step is to set up the neural network and to **randomly initialize the values of the weights**. And we usually initialize the weights to small values near zero.
  - Then we **implement forward propagation** so that we can input any excellent neural network and **compute  $h(x)$**  which is this **output vector of the  $y$  values**.
  - We then also **implement code to compute** this **cost function  $J(\Theta)$** .
  - And next we **implement back-prop**, or the back-propagation algorithm, to **compute these partial derivatives terms, partial derivatives of  $J(\Theta)$  with respect to the parameters**.

→ for  $i = 1:m$  {  $(x^{(1)}, y^{(1)})$   $(x^{(2)}, y^{(2)})$ , ...,  $(x^{(m)}, y^{(m)})$  }  
 → Perform forward propagation and backpropagation using example  $(x^{(i)}, y^{(i)})$   
 (Get activations  $a^{(l)}$  and delta terms  $\delta^{(l)}$  for  $l = 2, \dots, L$ ).  
 $\Delta^{(2)} := \Delta^{(2)} + \delta^{(1)}(a^{(1)})^T$   
 ...  
 compute  $\frac{\partial}{\partial \Theta_{ik}^{(2)}} J(\Theta)$ .

Concretely, to implement back prop. Usually we will do that with a for loop over the training examples. Some of you may have heard of advanced, and frankly very advanced factorization methods where you don't have a for loop over the m-training examples, that the first time you're implementing back prop there should almost certainly be for loop in your code, where you're iterating over the examples, you know,  $x^{(1)}, y^{(1)}$ , then so you do forward prop and back prop on the first example, and then in the second iteration of the for loop you do forward propagation and back propagation on the second example, and so on. Until you get through the final example. So there should be a for loop in your implementation of back prop, at least the first time implementing it.

And then there are frankly somewhat complicated ways to do this without a for loop, but I definitely do not recommend trying to do that much more complicated version the first time you try to implement back prop. So concretely, we have a for loop over my m-training examples and inside the for loop we're going to perform forward prop and back prop using just this one example. And what that means is that we're going to take  $x^{(i)}$  and feed that to my input layer, perform forward-prop, perform back-prop and that will give me all of these activations and all of these delta terms for all of the layers of all my units in the neural network, then still inside this for loop, let me draw some curly braces just to show the scope of the for loop, this is an octave code of course, but it's more a sequence Java code, and a for loop encompasses all this. We're going to compute those delta terms, which are, here is the formula that we gave earlier. Plus, you know, delta 1 plus one times a, 1 transpose and some other the code. And then finally, outside the for loop, you know, having computed these delta terms, these accumulation terms, we would then have some other code and then that will allow us to compute these partial derivative terms. Right and these partial derivative terms have to take into account the regularization term lambda as well. And so, those formulas were given in an earlier video.

So, having done that you now hopefully have code to compute these partial derivative terms. Next is step five, what I do is then use gradient checking to compare these partial derivative terms that were computed. So, I've compared the versions computed using back propagation versus the partial derivatives computed using the numerical estimates as using numerical estimates of the derivatives. So, I do gradient checking to make sure that both of these give you very similar values.

Having done gradient checking just now reassures us that our implementation of back propagation is correct, and is then very important that we disable gradient checking, because the gradient checking code is computationally very slow.

And finally, we then use an optimization algorithm such as **gradient descent**, or one of the advanced optimization methods such as LB GS, contract gradient as embodied into **fminunc** or other optimization methods. We use these together with back propagation, so back propagation is the thing that computes these partial derivatives for us. And so, we know how to compute the cost function, we know how to compute the partial derivatives using back propagation, so we can use one of these optimization methods to try to minimize  $J(\Theta)$  as a function of the parameters  $\Theta$ .

### Training a neural network

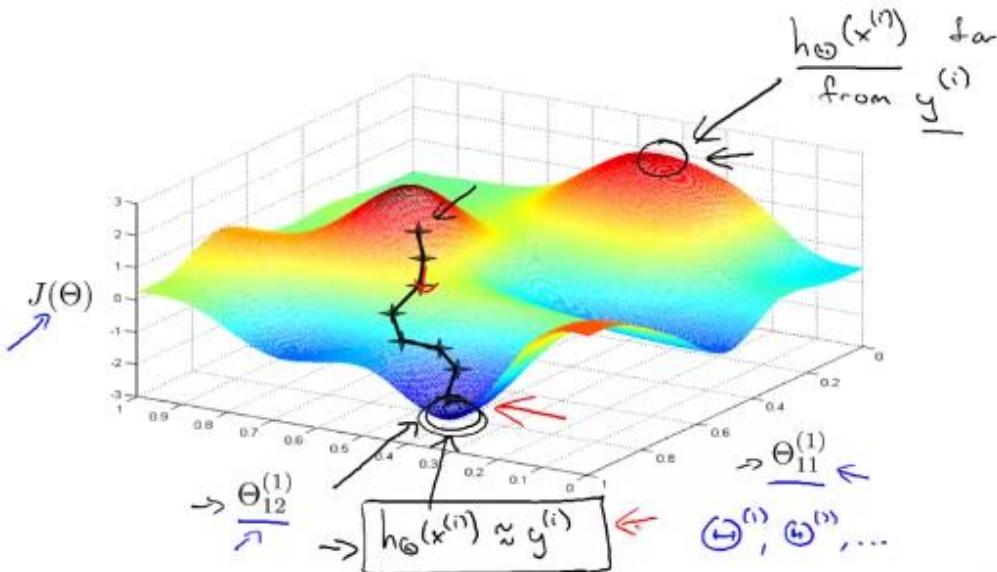
- 5. Use gradient checking to compare  $\frac{\partial}{\partial \Theta_{ik}^{(l)}} J(\Theta)$  computed using backpropagation vs. using numerical estimate of gradient of  $J(\Theta)$ .  
→ Then disable gradient checking code.
- 6. Use gradient descent or advanced optimization method with backpropagation to try to minimize  $J(\Theta)$  as a function of parameters  $\Theta$

$$\frac{\partial}{\partial \Theta_{ik}^{(l)}} J(\Theta)$$

And by the way, **for neural networks**, this **cost function  $J(\Theta)$  is non-convex**, or is not convex and so it can theoretically be **susceptible to local minima**, and in fact algorithms like gradient descent and the advance optimization methods can, in theory, get stuck in local optima, but it turns out that in practice this is not usually a huge problem and even though we can't guarantee that these algorithms will find a global optimum, usually algorithms like gradient descent will do a very good job minimizing this cost function  $J(\Theta)$  and get a very good local minimum, even if it doesn't get to the global optimum.

$$J(\Theta) - \text{non-convex.}$$

Finally, gradient descents for a neural network might still seem a little bit magical. So, let me just show one more figure to try to get that intuition about what gradient descent for a neural network is doing.



This was actually similar to the figure that I was using earlier to explain gradient descent. So, we have some cost function, and we have a number of parameters in our neural network. Right, here I've just written down two of the parameter values. In reality, of course, in a neural network, we can have lots of **parameters** with these  $\Theta^{(1)}$ ,  $\Theta^{(2)}$  all of these are **matrices**, right? So we can have very high dimensional parameters but because of the limitations the sorts of plots we can draw, I'm pretending that we have only two parameters in this neural network. Although obviously we have a lot more in practice.

Now, this cost function  $J(\Theta)$  measures how well the neural network fits the training data. So, if you take a point like this one, down here, that's a point where  $J(\Theta)$  is pretty low, and so this corresponds to a setting of the parameters. There's a setting of the parameters  $\Theta$ , where, you know, for most of the training examples, the output of my hypothesis that may be pretty close to  $y^{(i)}$ , and if this is true than that's what causes my cost function to be pretty low. Whereas in contrast, if you were to take a value like that, a point like that corresponds to, where for many training examples, the output of my neural network is far from the actual value  $y^{(i)}$  that was observed in the training set.

So points like this on the right correspond to where the hypothesis, where the neural network is outputting values on the training set that are far from  $y^{(i)}$ . So, it's not fitting the training set well, whereas points like this with low values of the cost function corresponds to where  $J(\Theta)$  is low, and therefore corresponds to where the neural network happens to be fitting my training set well, because I mean this is what's needed to be true in order for  $J(\Theta)$  to be small.

So what gradient descent does is we'll start from some random initial point like that one over there, and it will repeatedly go downhill. And so what back propagation is doing is computing the direction of the gradient, and what gradient descent is doing is it's taking little steps downhill until hopefully it gets to, in this case, a pretty good local optimum. So, when you implement back propagation and use gradient descent or one of the advanced optimization methods, this picture sort of explains what the algorithm is doing. It's trying to find a value of the parameters

where the output values in the neural network closely matches the values of the  $y^{(i)}$ 's observed in your training set.

So, hopefully this gives you a better sense of how the many different pieces of neural network learning fit together. In case even after this video, in case you still feel like there are, like, a lot of different pieces and it's not entirely clear what some of them do or how all of these pieces come together, that's actually okay.

Neural network learning and back propagation is a complicated algorithm. And even though I've seen the math behind back propagation for many years and I've used back propagation, I think very successfully, for many years, even today I still feel like I don't always have a great grasp of exactly what back propagation is doing sometimes. And what the optimization process looks like of minimizing  $J(\theta)$ .

This is a much harder algorithm to feel like I have a much less good handle on exactly what this is doing compared to say, linear regression or logistic regression. Which were mathematically and conceptually much simpler and much cleaner algorithms. But so in case if you feel the same way, you know, that's actually perfectly okay, but if you do implement back propagation, hopefully what you find is that this is one of the most powerful learning algorithms and if you implement this algorithm, implement back propagation, implement one of these optimization methods, you find that back propagation will be able to fit very complex, powerful, non-linear functions to your data, and this is one of the most effective learning algorithms we have today.

## Summary - Putting it Together

First, pick a network architecture; choose the layout of your neural network, including how many hidden units in each layer and how many layers in total you want to have.

- Number of input units = dimension of features  $x^{(i)}$
- Number of output units = number of classes
- Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units)
- Defaults: 1 hidden layer. If you have more than 1 hidden layer, then it is recommended that you have the same number of units in every hidden layer.

## Training a Neural Network

1. Randomly initialize the weights
2. Implement forward propagation to get  $h_{\theta}(x^{(i)})$  for any  $x^{(i)}$
3. Implement the cost function
4. Implement backpropagation to compute partial derivatives
5. Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.
6. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.

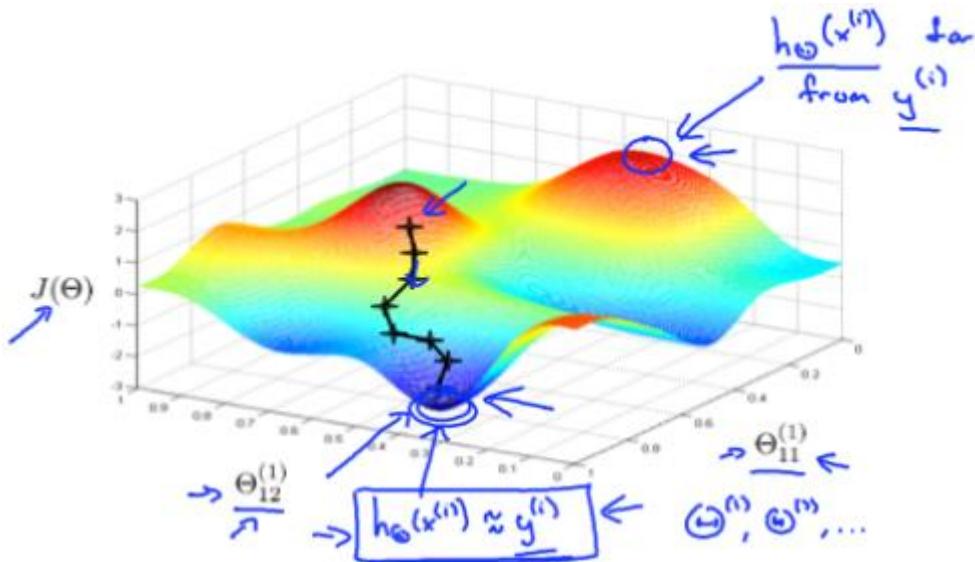
When we perform forward and back propagation, we loop on every training example:

```

1   for i = 1:m,
2       Perform forward propagation and backpropagation using example (x(i),
3           ),y(i))
4       (Get activations a(1) and delta terms d(l) for l = 2,...,L)

```

The following image gives us an intuition of what is happening as we are implementing our neural network:



Ideally, you want  $h_{\Theta}(x^{(i)}) \approx y^{(i)}$ . This will minimize our cost function. However, keep in mind that  $J(\Theta)$  is not convex and thus we can end up in a local minimum instead.

## Deciding What to Try Next

By now you have seen a lot of different learning algorithms.

And if you've been following along these videos you should consider yourself an expert on many state-of-the-art machine learning techniques.

But even among people that know a certain learning algorithm, there's often a huge difference between someone that really knows how to powerfully and effectively apply that algorithm, versus someone that's less familiar with some of the material that I'm about to teach and who doesn't really understand how to apply these algorithms and can end up wasting a lot of their time trying things out that don't really make sense.

What I would like to do is make sure that if you are developing machine learning systems, that you know how to choose one of the most promising avenues to spend your time pursuing. And on this and the next few videos I'm going to give a number of practical suggestions, advice, guidelines, on how to do that. And concretely what we'd focus on is the problem of, suppose you

are developing a machine learning system or trying to improve the performance of a machine learning system, how do you go about deciding what are the promising avenues to try next?

To explain this, let's continue using our example of learning to predict housing prices. And let's say you've implemented and regularized linear regression.

### **Debugging a learning algorithm:**

Suppose you have implemented regularized linear regression to predict housing prices.

$$\rightarrow J(\theta) = \frac{1}{2m} \left[ \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^m \theta_j^2 \right]$$

However, when you test your hypothesis on a new set of houses, you find that it makes unacceptably large errors in its predictions. What should you try next?

Thus minimizing that cost function  $J(\theta)$ . Now suppose that after you take your learning parameters, if you test your hypothesis on the new set of houses, suppose you find that this is making huge errors in this prediction of the housing prices.

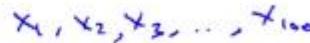
The question is what should you then try next in order to improve the learning algorithm?

There are many things that one can think of that could improve the performance of the learning algorithm.

One thing they could try, is to get more training examples. And concretely, you can imagine, maybe, you know, setting up phone surveys, going door to door, to try to get more data on how much different houses sell for.

And the sad thing is I've seen a lot of people spend a lot of time collecting more training examples, thinking oh, if we have twice as much or ten times as much training data, that is certainly going to help, right? But sometimes getting more training data doesn't actually help and in the next few videos we will see why, and we will see how you can avoid spending a lot of time collecting more training data in settings where it is just not going to help.

Other things you might try are to well maybe try a smaller set of features. So if you have some set of features such as  $x_1, x_2, x_3$  and so on, maybe a large number of features. Maybe you want to spend time carefully selecting some small subset of them to prevent overfitting.

- - Get more training examples
- Try smaller sets of features 
- - Try getting additional features
- Try adding polynomial features  $(x_1^2, x_2^2, x_1x_2, \text{etc.})$
- Try decreasing  $\lambda$
- Try increasing  $\lambda$

Or maybe you need to get additional features. Maybe the current set of features aren't informative enough and you want to collect more data in the sense of getting more features.

And once again this is the sort of project that can scale up to be a huge project you can imagine getting phone surveys, to find out more houses, or extra land surveys to find out more about the

pieces of land and so on, so a huge project. And once again it would be nice to know in advance if this is going to help before we spend a lot of time doing something like this.

We can also try adding polynomial features things like  $x_1$  square  $x_2$  square and product features  $x_1 x_2$ . We can still spend quite a lot of time thinking about that and we can also try other things like decreasing lambda, the regularization parameter or increasing lambda. Given a menu of options like these, some of which can easily scale up to six month or longer projects.

Unfortunately, the most common method that people use to pick one of these is to go by gut feeling. In which what many people will do is sort of randomly pick one of these options and maybe say, "Oh, let's go and get more training data." And easily spend six months collecting more training data or maybe someone else would rather be saying, "Well, let's go collect a lot more features on these houses in our data set." And I have a lot of times, sadly seen people spend, you know, literally 6 months doing one of these avenues that they have picked so much at random only to discover six months later that that really wasn't a promising avenue to pursue.

Fortunately, there is a pretty simple technique that can let you very quickly rule out half of the things on this list as being potentially promising things to pursue. And there is a very simple technique, that if you run, can easily rule out many of these options, and potentially save you a lot of time pursuing something that's just not going to work.

In the next two videos after this, I'm going to first talk about how to evaluate learning algorithms. And in the next few videos after that, I'm going to talk about these techniques, which are called the machine learning diagnostics. And what a diagnostic is, is a test you can run, to get insight into what is or isn't working with an algorithm, and which will often give you insight as to what are promising things to try to improve a learning algorithm's performance.

### **Machine learning diagnostic:**

**Diagnostic:** A test that you can run to gain insight what is/isn't working with a learning algorithm, and gain guidance as to how best to improve its performance.

**Diagnostics can take time to implement, but doing so can be a very good use of your time.**

We'll talk about specific diagnostics later in this video sequence. But I should mention in advance that diagnostics can take time to implement and can sometimes, you know, take quite a lot of time to implement and understand but doing so can be a very good use of your time when you are developing learning algorithms because they can often save you from spending many months pursuing an avenue that you could have found out much earlier just was not going to be fruitful.

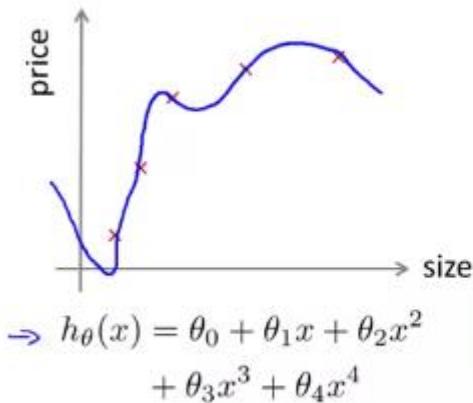
So in the next few videos, I'm going to first talk about how to evaluate your learning algorithms and after that I'm going to talk about some of these diagnostics which will hopefully let you much more effectively select more of the useful things to try to mix if your goal is to improve the performance of your machine learning system.

## Evaluating a Hypothesis

In this video, I would like to talk about how to evaluate a hypothesis that has been learned by your algorithm.

In later videos, we will build on this to talk about how to prevent in the problems of overfitting and underfitting as well. When we fit the parameters of our learning algorithm we think about choosing the parameters to minimize the training error.

### Evaluating your hypothesis



Fails to generalize to new examples not in training set.

$x_1$	= size of house
$x_2$	= no. of bedrooms
$x_3$	= no. of floors
$x_4$	= age of house
$x_5$	= average income in neighborhood
$x_6$	= kitchen size
:	
$x_{100}$	



One might think that getting a really low value of training error might be a good thing, but we have already seen that just because a hypothesis has low training error, that doesn't mean it is necessarily a good hypothesis. And we've already seen the example of how a hypothesis can overfit. And therefore fail to generalize the new examples not in the training set. So how do you tell if the hypothesis might be overfitting.

In this simple example we could plot the hypothesis  $h(x)$  and just see what was going on. But in general for problems with more features than just one feature, for problems with a large number of features like these it becomes hard or may be impossible to plot what the hypothesis function looks like and so we need some other way to evaluate our hypothesis.

The standard way to evaluate a learning hypothesis is as follows. Suppose we have a data set like this.

## Evaluating your hypothesis

Dataset:

Size	Price
2104	400
1600	330
2400	369
1416	232
3000	540
1985	300
1534	315
<hr/>	
1427	199
1380	212
1494	243

70% Training set →  $(x^{(1)}, y^{(1)})$ ,  $(x^{(2)}, y^{(2)})$ , ...,  $(x^{(m)}, y^{(m)})$

30% Test set →  $(x_{test}^{(1)}, y_{test}^{(1)})$ ,  $(x_{test}^{(2)}, y_{test}^{(2)})$ , ...,  $(x_{test}^{(m_{test})}, y_{test}^{(m_{test})})$

$m_{test}$  = no. of test examples

Andrew

Here I have just shown 10 training examples, but of course usually we may have dozens or hundreds or maybe thousands of training examples. In order to make sure we can evaluate our hypothesis, what we are going to do is split the data we have into two portions. The first portion is going to be our usual training set and the second portion is going to be our test set, and a pretty typical split of this all the data we have into a training set and test set might be around say a 70%, 30% split. Worth more today to grade the training set and relatively less to the test set.

And so now, if we have some data set, we would assign only say 70% of the data to be our training set where here " $m$ " is as usual our number of training examples and the remainder of our data might then be assigned to become our test set. And here, I'm going to use the notation  $m$  subscript test to denote the number of test examples. And so in general, this subscript test is going to denote examples that come from a test set so that  $x_1$  subscript test,  $y_1$  subscript test is my first test example which I guess in this example might be this example over here.

Finally, one last detail whereas here I've drawn this as though the first 70% goes to the training set and the last 30% to the test set. If there is any sort of ordering to the data that should be better to send a random 70% of your data to the training set and a random 30% of your data to the test set. So if your data were already randomly sorted, you could just take the first 70% and last 30% that if your data were not randomly ordered, it would be better to randomly shuffle or to randomly reorder the examples in your training set before you know sending the first 70% in the training set and the last 30% of the test set.

Here then is a fairly typical procedure for how you would train and test a learning algorithm may be the learning regression. First, you learn the parameters  $\theta$  from the training set, so you minimize the usual training error objective  $J(\theta)$ , where  $J(\theta)$  here was defined using that 70% of all the data you have. There is only the training data.

## Training/testing procedure for linear regression

- - Learn parameter  $\theta$  from training data (minimizing training error  $J(\theta)$ ) 70%
- Compute test set error:

$$J_{\text{test}}(\theta) = \frac{1}{2m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} (h_{\theta}(x_{\text{test}}^{(i)}) - y_{\text{test}}^{(i)})^2$$

And then you would compute the test error. And I am going to denote the test error as  $J$  subscript test. And so what you do is take your parameter  $\theta$  that you have learned from the training set, and plug it in here and compute your test set error, which I am going to write as follows. So this is basically the average squared error as measured on your test set. It's pretty much what you'd expect.

So if we run every test example through your hypothesis with parameter  $\theta$  and just measure the squared error that your hypothesis has on your  $M$  subscript test, test examples. And of course, this is the definition of the test set error if we are using linear regression and using the squared error metric.

How about if we were doing a classification problem and say using logistic regression instead.

In that case, the procedure for training and testing say logistic regression is pretty similar first we will do the parameters from the training data, that first 70% of the data. And it will compute the test error as follows.

It's the same objective function as we always use but we just use logistic regression, except that now its defined using our  $M$  subscript test, test examples. While this definition of the test set error  $J$  subscript test is perfectly reasonable. Sometimes there is an alternative test sets metric that might be easier to interpret, and that's the misclassification error. It's also called the 0/1 misclassification error, with 0/1 denoting that you either get an example right or you get an example wrong. Here's what I mean.

## Training/testing procedure for logistic regression

- - Learn parameter  $\theta$  from training data  $m_{test}$
- Compute test set error:
- $$J_{test}(\theta) = -\frac{1}{m_{test}} \sum_{i=1}^{m_{test}} y_{test}^{(i)} \log h_{\theta}(x_{test}^{(i)}) + (1 - y_{test}^{(i)}) \log h_{\theta}(x_{test}^{(i)})$$
- Misclassification error (0/1 misclassification error):
 
$$\text{err}(h_{\theta}(x), y) = \begin{cases} 1 & \text{if } h_{\theta}(x) \geq 0.5, y = 0 \\ & \text{or if } h_{\theta}(x) < 0.5, y = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\text{Test error} = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} \text{err}(h_{\theta}(x_{test}^{(i)}), y_{test}^{(i)}).$$

Let me define the error of a prediction. That is  $h(x)$  and given the label  $y$  as equal to one if my hypothesis outputs the value greater than or equal to five and  $Y$  is equal to zero, or if my hypothesis outputs a value of less than 0.5 and  $y$  is equal to one, right, so both of these cases basically respond to if your hypothesis mislabeled the example assuming your threshold is at 0.5. So either though it was more likely to be 1, but it was actually 0, or your hypothesis stored was more likely to be 0, but the label was actually 1. And otherwise, we define this error function to be zero if your hypothesis basically classified the example  $y$  correctly. We could then define the test error, using the misclassification error metric to be one over  $M$  tests of sum from  $i$  equals one to  $M$  subscript test of the error of  $h(x(i))$  test comma  $y(i)$ . And so that's just my way of writing out that this is exactly the fraction of the examples in my test set that my hypothesis has mislabeled. And so that's the definition of the test set error using the misclassification error of the 0/1 misclassification error metric.

So that's the standard technique for evaluating how good a learned hypothesis is. In the next video, we will adapt these ideas to helping us do things like choose what features like the degree polynomial to use with the learning algorithm or choose the regularization parameter for learning algorithm.

## Summary: Evaluating a Hypothesis

Once we have done some trouble shooting for errors in our predictions by:

- Getting more training examples
- Trying smaller sets of features
- Trying additional features
- Trying polynomial features
- Increasing or decreasing  $\lambda$

We can move on to evaluate our new hypothesis.

A hypothesis may have a low error for the training examples but still be inaccurate (because of overfitting). Thus, to evaluate a hypothesis, given a dataset of training examples, we can split up the data into two sets: a **training set** and a **test set**. Typically, the training set consists of 70 % of your data and the test set is the remaining 30 %.

The new procedure using these two sets is then:

1. Learn  $\theta$  and minimize  $J_{\text{train}}(\theta)$  using the training set
2. Compute the test set error  $J_{\text{test}}(\theta)$

## The test set error

1. For linear regression:  $J_{\text{test}}(\theta) = \frac{1}{2m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} (h_{\theta}(x_{\text{test}}^{(i)}) - y_{\text{test}}^{(i)})^2$

2. For classification ~ Misclassification error (aka 0/1 misclassification error):

$$\text{err}(h_{\theta}(x), y) = \begin{cases} 1 & \text{if } h_{\theta}(x) \geq 0.5 \text{ and } y = 0 \text{ or } h_{\theta}(x) < 0.5 \text{ and } y = 1 \\ 0 & \text{otherwise} \end{cases}$$

This gives us a binary 0 or 1 error result based on a misclassification. The average test error for the test set is:

$$\text{Test Error} = \frac{1}{m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} \text{err}(h_{\theta}(x_{\text{test}}^{(i)}), y_{\text{test}}^{(i)})$$

This gives us the proportion of the test data that was misclassified.

## Model Selection and Train/Validation/Test Sets

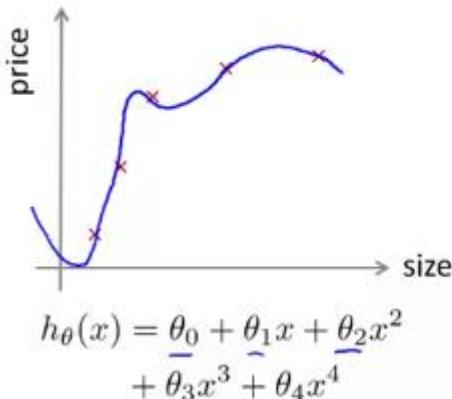
Suppose you would like to decide what degree of polynomial to fit to a data set. So that what features to include that gives you a learning algorithm. Or suppose you'd like to choose the regularization parameter lambda for learning algorithm.

How do you do that?

These are called model selection problems. And in our discussion of how to do this, we'll talk about not just how to split your data into train and test sets, but how to switch data into what we discover is called the train, validation, and test sets. We'll see in this video just what these things are, and how to use them to do model selection.

We've already seen a lot of times the problem of overfitting, in which just because a learning algorithm fits a training set well, that doesn't mean it's a good hypothesis. More generally, this is why the training set error is not a good predictor for how well the hypothesis will do on new example.

## Overfitting example



Once parameters  $\theta_0, \theta_1, \dots, \theta_4$  were fit to some set of data (training set), the error of the parameters as measured on that data (the training error  $J(\theta)$ ) is likely to be lower than the actual generalization error.

Concretely, if you fit some set of parameters. Theta0, theta1, theta2, and so on, to your training set, then the fact that your hypothesis does well on the training set, well, this doesn't mean much in terms of predicting how well your hypothesis will generalize to new examples not seen in the training set. And a more general principle is that once your parameter is what fit to some set of data, may be the training set, may be something else, then the error of your hypothesis as measured on that same data set, such as the training error, that's unlikely to be a good estimate of your actual generalization error, that is how well the hypothesis will generalize to new examples.

Now let's consider the model selection problem. Let's say you're trying to choose what degree polynomial to fit to data.

### Model selection

- $\rightarrow d = \text{degree of polynomial}$
- $d=1$  1.  $\rightarrow h_{\theta}(x) = \theta_0 + \theta_1 x \rightarrow \Theta^{(1)} \rightarrow J_{test}(\Theta^{(1)})$
  - $d=2$  2.  $\rightarrow h_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 \rightarrow \Theta^{(2)} \rightarrow J_{test}(\Theta^{(2)})$
  - $d=3$  3.  $\rightarrow h_{\theta}(x) = \theta_0 + \theta_1 x + \dots + \theta_3 x^3 \rightarrow \Theta^{(3)} \rightarrow J_{test}(\Theta^{(3)})$
  - $\vdots$
  - $d=10$  10.  $\rightarrow h_{\theta}(x) = \theta_0 + \theta_1 x + \dots + \theta_{10} x^{10} \rightarrow \Theta^{(10)} \rightarrow J_{test}(\Theta^{(10)})$

Choose  $\boxed{\theta_0 + \dots + \theta_5 x^5}$

How well does the model generalize? Report test set error  $J_{test}(\theta^{(5)})$ .

Problem:  $J_{test}(\theta^{(5)})$  is likely to be an optimistic estimate of generalization error. I.e. our extra parameter ( $d = \text{degree of polynomial}$ ) is fit to test set.

So, should you choose a linear function, a quadratic function, a cubic function? All the way up to a 10th-order polynomial. So it's as if there's one extra parameter in this algorithm, which I'm going to denote  $d$ , which is, what degree of polynomial do you want to pick. So it's as if, in addition to the  $\theta$  parameters, it's as if there's one more parameter,  $d$ , that you're trying to determine using your data set.

So, the first option is  $d$  equals one, if you fit a linear function. We can choose  $d$  equals two,  $d$  equals three, all the way up to  $d$  equals 10. So, we'd like to fit this extra sort of parameter which I'm denoting by  $d$ . And concretely let's say that you want to choose a model that is choose a degree of polynomial, choose one of these 10 models, and fit that model and also get some estimate of how well your fitted hypothesis will generalize to new examples.

Here's one thing you could do. What you could, first take your first model and minimize the training error, and this would give you some parameter vector  $\theta$ . And you could then take your second model, the quadratic function, and fit that to your training set and this will give you some other parameter vector  $\theta$ .

In order to distinguish between these different parameter vectors, I'm going to use a superscript one superscript two there where  $\theta$  superscript one just means the parameters I get by fitting this model to my training data. And  $\theta$  superscript two just means the parameters I get by fitting this quadratic function to my training data and so on.

By fitting a cubic model I get parenthesis three up to, well, say  $\theta^{(10)}$ . And one thing we could do is then take these parameters and look at the test set error. So I can compute on my test set  $J_{\text{test}}(\theta^{(1)})$ ,  $J_{\text{test}}(\theta^{(2)})$ , and so on.  $J_{\text{test}}(\theta^{(3)})$ , and so on. So I'm going to take each of my hypotheses with the corresponding parameters and just measure the performance of on the test set.

Now, one thing I could do then is, in order to select one of these models, I could then see which model has the lowest test set error. And let's just say for this example that I ended up choosing the fifth order polynomial.

So, this seems reasonable so far. But now let's say I want to take my fifth hypothesis, this, this, fifth order model, and let's say I want to ask, how well does this model generalize? One thing I could do is look at how well my fifth order polynomial hypothesis had done on my test set. But the problem is this will not be a fair estimate of how well my hypothesis generalizes. And the reason is what we've done is we've fit this extra parameter  $d$ , that is this degree of polynomial. And what fits that parameter  $d$ , using the test set, namely, we chose the value of  $d$  that gave us the best possible performance on the test set.

And so, the performance of my parameter vector  $\theta^{(5)}$ , on the test set, that's likely to be an overly optimistic estimate of generalization error. Right, so, that because I had fit this parameter  $d$  to my test set is no longer fair to evaluate my hypothesis on this test set, because I fit my parameters to this test set, I've chose the degree  $d$  of polynomial using the test set. And so my hypothesis is likely to do better on this test set than it would on new examples that it hasn't seen before, and that's which is, which is what I really care about.

So just to reiterate, on the previous slide, we saw that if we fit some set of parameters, you know, say  $\theta_0, \theta_1$ , and so on, to some training set, then the performance of the fitted model on the training set is not predictive of how well the hypothesis will generalize to new examples is because these parameters were fit to the training set, so they're likely to do well on the training set, even if the parameters don't do well on other examples. And, in the procedure I just

described on this slide, we just did the same thing. And specifically, what we did was, we fit this parameter  $\theta$  to the test set. And by having fit the parameter to the test set, this means that the performance of the hypothesis on that test set may not be a fair estimate of how well the hypothesis is, is likely to do on examples we haven't seen before.

To address this problem, in a model selection setting, if we want to evaluate a hypothesis, this is what we usually do instead.

### Train/validation/test error

Training error:

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

Cross Validation error:

$$\rightarrow J_{cv}(\theta) = \frac{1}{2m_{cv}} \sum_{i=1}^{m_{cv}} (h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$$

Test error:

$$J_{test}(\theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} (h_\theta(x_{test}^{(i)}) - y_{test}^{(i)})^2$$

Given the data set, instead of just splitting into a training test set, what we're going to do is then split it into three pieces. And the first piece is going to be called the training set as usual. So let me call this first part the training set. And the second piece of this data, I'm going to call the cross validation set. Cross validation. And I am going to abbreviate cross validation as (CV). Sometimes it's also called the validation set instead of cross validation set.

And then the last column I am going to call my usual test set. And the pretty, pretty typical ratio at which to split these things will be to send 60% of your data's, your training set, maybe 20% to your cross validation set, and 20% to your test set. And these numbers can vary a little bit but this distribution is very typical.

And so our training sets will now be only maybe 60% of the data, and our cross-validation set, or our validation set, will have some number of examples, I'm going to denote that  $M_{cv}$ . So that's the number of cross-validation examples. Following our early notational convention I'm going to use  $x_{cv}^{(i)}$ ,  $y_{cv}^{(i)}$ , to denote the  $i^{\text{th}}$  cross validation example. And finally we also have a test set over here with our  $M_{test}$  being the number of test examples.

So, now that we've defined the training validation or cross validation and test sets. We can also define the training error, cross validation error, and test error. So here's my training error, and I'm just writing this as  $J_{train}(\theta)$ . This is pretty much the same things. These are the same thing as the  $J(\theta)$  that I've been writing so far, this is just a training set error you know, as measured on your training set, and then  $J_{cv}(\theta)$  my cross validation error, this is pretty much what you'd expect, just like the training error you've set measure on a cross validation data set, and here's my test set error same as before.

So when faced with a model selection problem like this, what we're going to do is, instead of using the test set to select the model, we're instead going to use the validation set, or the cross validation set, to select the model.

### Model selection

$$\begin{aligned}
 & \text{d=1} \quad 1. \quad h_{\theta}(x) = \theta_0 + \theta_1 x \rightarrow \min_{\theta} J(\theta) \rightarrow \theta^{(1)} \rightarrow J_{cv}(\theta^{(1)}) \\
 & \text{d=2} \quad 2. \quad h_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 \rightarrow \theta^{(2)} \rightarrow J_{cv}(\theta^{(2)}) \\
 & \text{d=3} \quad 3. \quad h_{\theta}(x) = \theta_0 + \theta_1 x + \dots + \theta_3 x^3 \rightarrow \theta^{(3)} \rightarrow J_{cv}(\theta^{(3)}) \\
 & \vdots \qquad \vdots \\
 & \text{d=10} \quad 10. \quad h_{\theta}(x) = \theta_0 + \theta_1 x + \dots + \theta_{10} x^{10} \rightarrow \theta^{(10)} \rightarrow J_{cv}(\theta^{(10)})
 \end{aligned}$$

$d=4$

Pick  $\theta_0 + \theta_1 x_1 + \dots + \theta_4 x^4$

Estimate generalization error for test set  $J_{test}(\theta^{(4)})$

Concretely, we're going to first take our first hypothesis, take this first model, and say, minimize the cross function, and this would give me some parameter vector  $\theta$  for the linear model. And, as before, I'm going to put a superscript 1, just to denote that this is the parameter for the linear model. We do the same thing for the quadratic model. Get some parameter vector  $\theta^{(2)}$ .

Get some para, parameter vector  $\theta^{(3)}$ , and so on, down to  $\theta^{(10)}$  for the polynomial. And what I'm going to do is, instead of testing these hypotheses on the test set, I'm instead going to test them on the cross validation set. And measure  $J_{cv}$ , to see how well each of these hypotheses do on my cross validation set. And then I'm going to pick the hypothesis with the lowest cross validation error. So for this example, let's say for the sake of argument, that it was my 4th order polynomial that had the lowest cross validation error. So in that case I'm going to pick this fourth order polynomial model. And finally, what this means is that that parameter  $d$ , remember  $d$  was the degree of polynomial, right? So  $d$  equals two,  $d$  equals three, all the way up to  $d$  equals 10. What we've done is we'll fit that parameter  $d$  and we'll say  $d$  equals four. And we did so using the cross-validation set. And so this degree of polynomial, so the parameter, is no longer fit to the test set, and so we've not saved away the test set, and we can use the test set to measure, or to estimate the generalization error of the model that was selected by this algorithm.

So, that was model selection and how you can take your data, split it into a training, validation, and test set. And use your cross validation data to select the model and evaluate it on the test set. One final note, I should say that in the machine learning, as is this practice today, there aren't many people that will do that earlier thing that I talked about, and said that you know, it isn't such a good idea, of selecting your model using the test set and then using the same test set to report the error as though selecting your degree of polynomial on the test set, and then reporting the error on the test set as though that were a good estimate of generalization error.

That sort of practice is unfortunately many many people do it. If you have a massive, massive test that is maybe not a terrible thing to do, but many practitioners, most practitioners, that machine learning tend to advise against that. And it's considered better practice to have separate train, validation and test sets.

As it turns out sometimes people do, you know, use the same data for the purpose of the validation set, and for the purpose of the test set. You only have a training set and a test set, that's considered that's a good practice, and you will see some people do it. But, if possible, I would recommend against doing that yourself.

## Summary: Model Selection and Train/Validation/Test Sets

Just because a learning algorithm fits a training set well, that does not mean it is a good hypothesis. It could over fit and as a result your predictions on the test set would be poor. The error of your hypothesis as measured on the data set with which you trained the parameters will be lower than the error on any other data set.

Given many models with different polynomial degrees, we can use a systematic approach to identify the 'best' function. In order to choose the model of your hypothesis, you can test each degree of polynomial and look at the error result.

One way to break down our dataset into the three sets is:

- Training set: 60%
- Cross validation set: 20%
- Test set: 20%

We can now calculate three separate error values for the three different sets using the following method:

1. Optimize the parameters in  $\theta$  using the training set for each polynomial degree.
2. Find the polynomial degree  $d$  with the least error using the cross validation set.
3. Estimate the generalization error using the test set with  $J_{\text{test}}(\theta^{(d)})$ , ( $d = \theta$  from polynomial with lower error);

This way, the degree of the polynomial  $d$  has not been trained using the test set.

## Bias vs. Variance

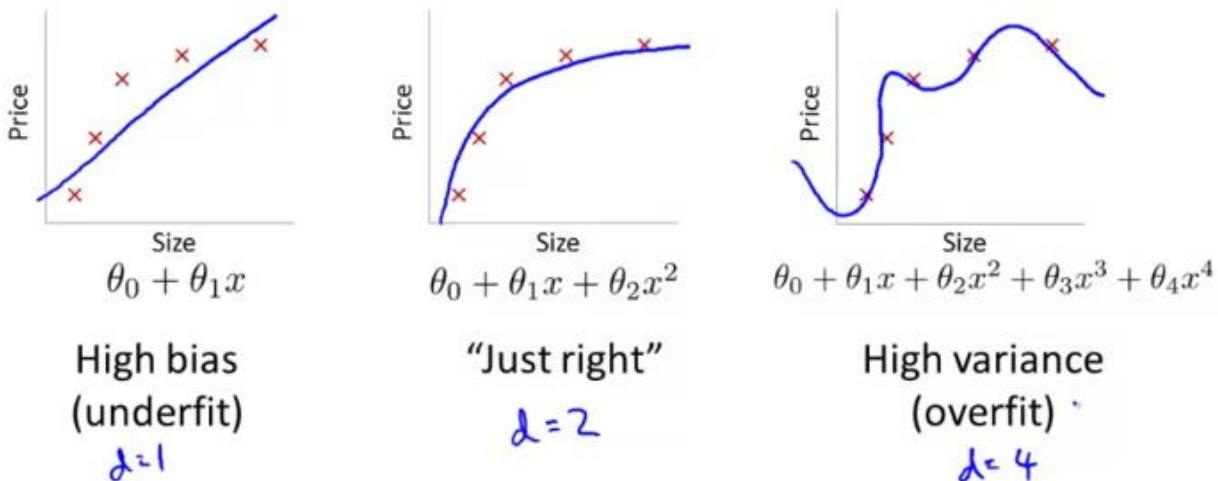
### Diagnosing Bias vs. Variance

If you run a learning algorithm and it doesn't do as well as you are hoping, almost all the time it will be because you have either a high bias problem or a high variance problem, in other words, either an underfitting problem or an overfitting problem.

And in this case, it's very important to figure out which of these two problems is bias or variance or a bit of both that you actually have. Because knowing which of these two things is happening would give a very strong indicator for what are the useful and promising ways to try to improve your algorithm.

In this video, I'd like to delve more deeply into this bias and variance issue and understand them better as well as figure out how to look into a learning algorithm and evaluate or diagnose whether we might have a bias problem or a variance problem since this will be critical to figuring out how to improve the performance of a learning algorithm that you will implement.

## Bias/variance



So, you've already seen this figure a few times where if you fit two simple hypothesis like a straight line that underfits the data, if you fit a two complex hypothesis, then that might fit the training set perfectly but overfit the data and this may be hypothesis of some intermediate level of complexities of some maybe degree two polynomials or not too low and not too high degree that's like just right and gives you the best generalization error over these options.

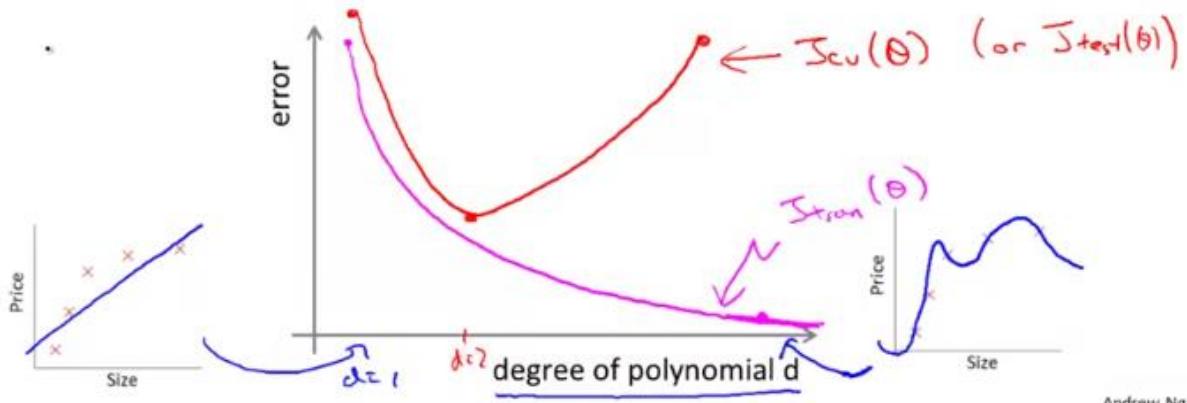
Now that we're armed with the notion of training and validation in test sets, we can understand the concepts of bias and variance a little bit better.

Concretely, let's let our training error and cross validation error be defined as in the previous videos. Just say the squared error, the average squared error, as measured on the training sets or as measured on the cross validation set.

Now, let's plot the following figure. On the horizontal axis I'm going to plot the degree of polynomial. So, as I go to the right I'm going to be fitting higher and higher order polynomials. So where the left of this figure where maybe  $d$  equals one, we're going to be fitting very simple functions whereas we're here on the right of the horizontal axis, I have much larger values of  $d$ , of a much higher degree polynomial. So here, that's going to correspond to fitting much more complex functions to your training set.

**Training error:**  $J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$

**Cross validation error:**  $J_{cv}(\theta) = \frac{1}{2m_{cv}} \sum_{i=1}^{m_{cv}} (h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$  (or  $J_{test}(\theta)$ )



Let's look at the training error and the cross validation error and plot them on this figure. Let's start with the training error.

As we increase the degree of the polynomial, we're going to be able to fit our training set better and better and so if  $d$  equals one, then there is high training error, if we have a very high degree of polynomial our training error is going to be really low, maybe even 0 because will fit the training set really well.

So, as we increase the degree of polynomial, we find typically that the training error decreases. So I'm going to write  $J_{train}(\theta)$  there, because our training error tends to decrease with the degree of the polynomial that we fit to the data.

Next, let's look at the cross-validation error or for that matter, if we look at the test set error, we'll get a pretty similar result as if we were to plot the cross validation error. So, we know that if  $d$  equals one, we're fitting a very simple function and so we may be under-fitting the training set and so it's going to be very high cross-validation error. If we fit an intermediate degree polynomial, we had  $d$  equals two in our example in the previous slide, we're going to have a much lower cross-validation error because we're finding a much better fit to the data.

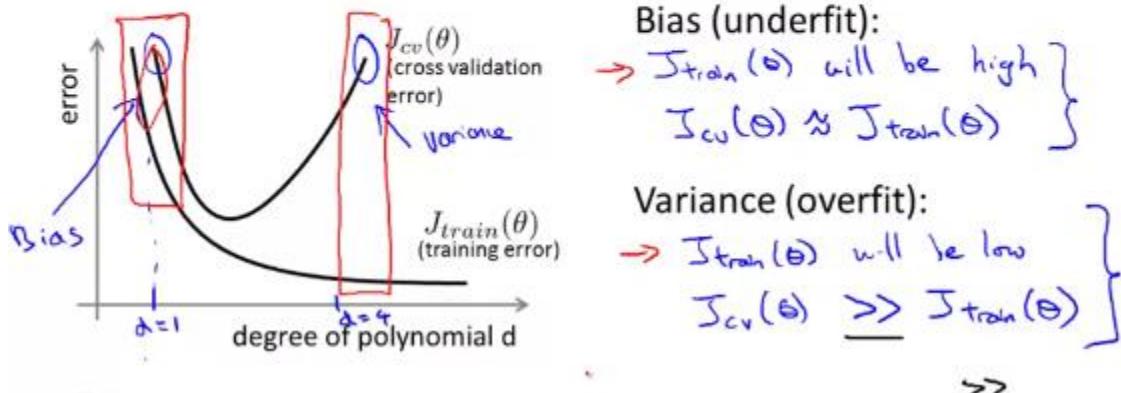
And conversely, if  $d$  were too high. So if  $d$  took on say a value of four, then we're again over-fitting, and so we end up with a high value for cross-validation error. So, if you were to vary this smoothly and plot a curve, you might end up with a curve like that where that's  $J_{cv}(\theta)$ . And again, if you plot  $J_{test}(\theta)$  you get something very similar. So, this sort of plot also helps us to better understand the notions of bias and variance.

Concretely, suppose you have applied a learning algorithm and it's not performing as well as you are hoping, so if your cross-validation set error or your test set error is high, how can we figure out if the learning algorithm is suffering from high bias or suffering from high variance?

So, the setting of a cross-validation error being high corresponds to either this regime or this regime.

### Diagnosing bias vs. variance

Suppose your learning algorithm is performing less well than you were hoping. ( $J_{cv}(\theta)$  or  $J_{test}(\theta)$  is high.) Is it a bias problem or a variance problem?



So, this regime on the left corresponds to a high bias problem. That is, if you are fitting a overly low order polynomial such as a d equals one when we really needed a higher order polynomial to fit to data. Whereas in contrast this regime corresponds to a high variance problem. That is, if d the degree of polynomial was too large for the data set that we have, and this figure gives us a clue for how to distinguish between these two cases. Concretely, for the high bias case, that is the case of underfitting, what we find is that both the cross validation error and the training error are going to be high. So, if your algorithm is suffering from a bias problem, the training set error will be high and you might find that the cross validation error will also be high. It might be close, maybe just slightly higher, than the training error. So, if you see this combination, that's a sign that your algorithm may be suffering from high bias.

In contrast, if your algorithm is suffering from high variance, then if you look here, we'll notice that  $J_{train}(\theta)$ , that is the training error, is going to be low. That is, you're fitting the training set very well, whereas your cross validation error assuming that this is, say, the squared error which we're trying to minimize say, whereas in contrast your error on a cross validation set or your cross function or cross validation set will be much bigger than your training set error. So, this is a double greater than sign, that's the math symbol for much greater than, denoted by two greater than signs. So if you see this combination of values, then that might be, that's a clue that your learning algorithm may be suffering from high variance and might be overfitting.

And the key that distinguishes these two cases is, if you have a high bias problem your training set error will also be high if your hypothesis just not fitting the training set well. And if you have a high variance problem, your training set error will usually be low, that is much lower than your cross-validation error.

So hopefully that gives you a somewhat better understanding of the two problems of bias and variance.

I still have a lot more to say about bias and variance in the next few videos, but what we'll see later is that by diagnosing whether a learning algorithm may be suffering from high bias or high variance, I'll show you even more details on how to do that in later videos. But we'll see that by figuring out whether a learning algorithm may be suffering from high bias or high variance or combination of both, that would give us much better guidance for what might be promising things to try in order to improve the performance of a learning algorithm.

## Summary: Diagnosing Bias vs Variance

In this section we examine the relationship between the degree of the polynomial  $d$  and the underfitting or overfitting of our hypothesis.

- We need to distinguish whether **bias** or **variance** is the problem contributing to bad predictions.
- High bias is underfitting and high variance is overfitting. Ideally, we need to find a golden mean between these two.

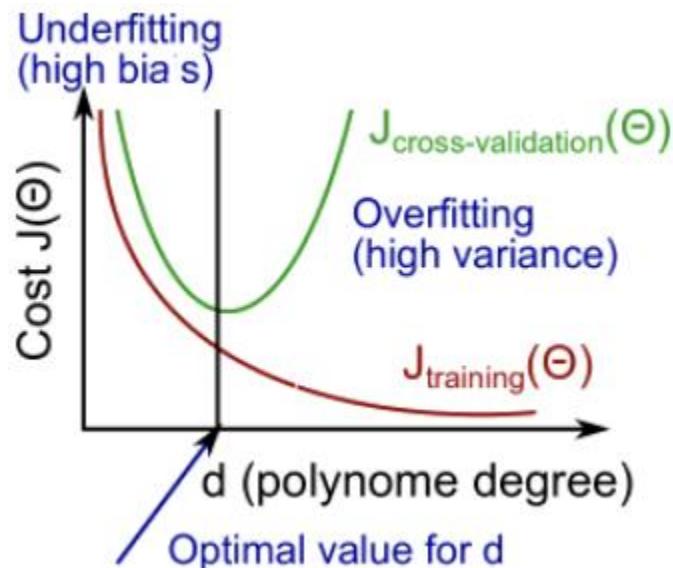
The training error will tend to **decrease** as we increase the degree  $d$  of the polynomial.

At the same time, the cross validation error will tend to **decrease** as we increase  $d$  up to a point, and then it will **increase** as  $d$  is increased, forming a convex curve.

**High bias (underfitting):** both  $J_{\text{train}}(\theta)$  and  $J_{\text{CV}}(\theta)$  will be high. Also,  $J_{\text{CV}}(\theta) \approx J_{\text{train}}(\theta)$ .

**High variance (overfitting):**  $J_{\text{train}}(\theta)$  will be low and  $J_{\text{CV}}(\theta)$  will be much greater than  $J_{\text{train}}(\theta)$ .

This is summarized in the figure below:



## Regularization and Bias/Variance

You've seen how regularization can help prevent over-fitting.

But how does it affect the bias and variances of a learning algorithm?

In this video I'd like to go deeper into the issue of bias and variance and talk about how it interacts with and is affected by the regularization of your learning algorithm.

Suppose we're fitting a high order polynomial, like that showed here, but to prevent over fitting we are going to use regularization, like that shown here. So we have this regularization term to try to keep the values of the parameters small. And as usual, the regularization comes from  $j = 1$  to  $m$ , rather than  $j = 0$  to  $m$ .

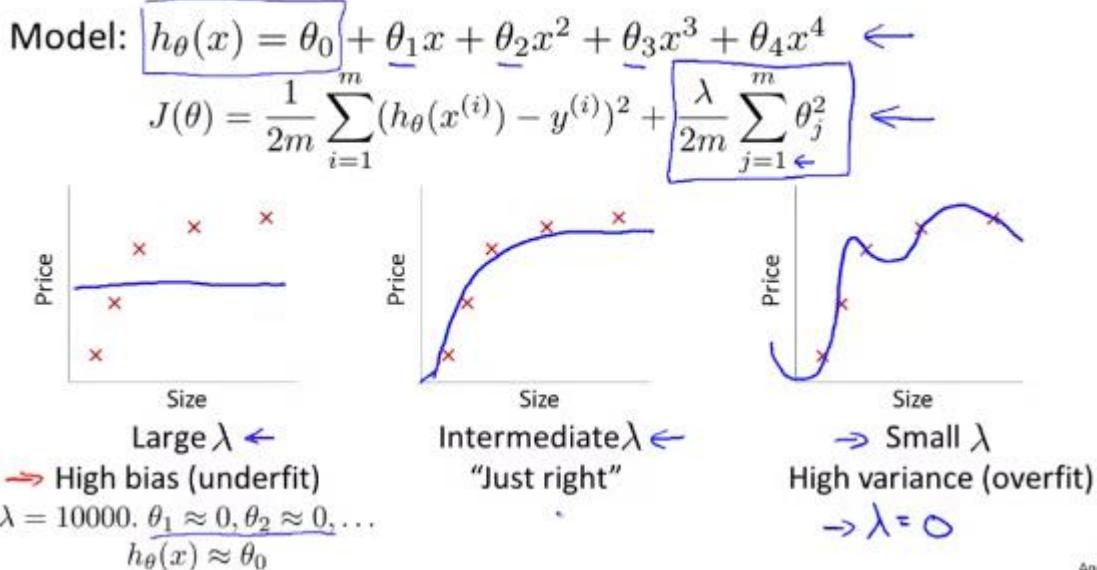
### Linear regression with regularization

$$\text{Model: } h_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4 \leftarrow$$

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + \boxed{\frac{\lambda}{2m} \sum_{j=1}^m \theta_j^2} \leftarrow$$

Let's consider three cases. The first is the case of a very large value of the regularization parameter lambda, such as if lambda were equal to 10,000. Some huge value. In this case, all of these parameters, theta 1, theta 2, theta 3, and so on would be heavily penalized and so we end up with most of these parameter values being closer to zero. And the hypothesis will be roughly  $h(x)$ , just equal or approximately equal to  $\theta_0$ .

### Linear regression with regularization



So we end up with a hypothesis that more or less looks like that, more or less a flat, constant straight line. And so this hypothesis has high bias and it badly under-fits this data set, so the horizontal straight line is just not a very good model for this data set.

At the other extreme is if we have a very small value of lambda, such as if lambda were equal to zero. In that case, given that we're fitting a high order polynomial, this is a usual over-fitting setting. In that case, given that we're fitting a high-order polynomial, basically, without regularization or with very minimal regularization, we end up with our usual high-variance, over fitting setting. This is basically if lambda is equal to zero, we're just fitting with our regularization, so that over fits the hypothesis. And it's only if we have some intermediate value of lambda that is neither too large nor too small that we end up with parameters  $\theta$  that give us a reasonable fit to this data.

So, how can we automatically choose a good value for the regularization parameter lambda?

### Choosing the regularization parameter $\lambda$

$$\begin{aligned} h_{\theta}(x) &= \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4 && \leftarrow \\ J(\theta) &= \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m} \sum_{j=1}^m \theta_j^2 && \leftarrow \\ \Rightarrow J_{train}(\theta) &= \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 && J(\theta) \end{aligned}$$

Just to reiterate, here's our model, and here's our learning algorithm's objective. For the setting where we're using regularization, let me define  $J_{train}(\theta)$  to be something different, to be the optimization objective, but without the regularization term. Previously, in an earlier video, when we were not using regularization I defined  $J_{train}(\theta)$  to be the same as  $J(\theta)$  as the cost function, but when we're using regularization, when there this extra lambda term, we're going to define  $J_{train}$  my training set error to be just my sum of squared errors on the training set or my average squared error on the training set without taking into account that regularization term.

And similarly I'm then also going to define the cross validation set error and to test that error as before to be the average sum of squared errors on the cross validation on the test sets. So just to summarize my definitions of  $J_{train}(\theta)$ ,  $J_{cv}(\theta)$ , and  $J_{test}(\theta)$  are just the average square error or one half of the average square error on my training validation of the test set without the extra regularization term.

$$\begin{aligned} \Rightarrow J_{train}(\theta) &= \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 && J(\theta) \\ J_{cv}(\theta) &= \frac{1}{2m_{cv}} \sum_{i=1}^{m_{cv}} (h_{\theta}(x_{cv}^{(i)}) - y_{cv}^{(i)})^2 && J \\ J_{test}(\theta) &= \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} (h_{\theta}(x_{test}^{(i)}) - y_{test}^{(i)})^2 && J \end{aligned}$$

So, this is how we can automatically choose the regularization parameter lambda. So what I usually do is maybe have some range of values of lambda I want to try out. So I might be considering not using regularization or here are a few values I might try lambda considering

$\lambda = 0.01, 0.02, 0.04$ , and so on. And I usually set these up in multiples of two, until some maybe larger value. If I were doing these in multiples of 2 I'd end up with a 10.24. It's 10 exactly, but this is close enough. And the three to four decimal places won't effect your result that much.

### Choosing the regularization parameter $\lambda$

Model:  $h_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

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

- 1. Try  $\lambda = 0$   $\rightarrow \min_{\theta} J(\theta) \rightarrow \theta^{(1)} \rightarrow J_{cv}(\theta^{(1)})$
  - 2. Try  $\lambda = 0.01$   $\rightarrow \min_{\theta} J(\theta) \rightarrow \theta^{(2)} \rightarrow J_{cv}(\theta^{(2)})$
  - 3. Try  $\lambda = 0.02$   $\rightarrow \theta^{(3)} \rightarrow J_{cv}(\theta^{(3)})$
  - 4. Try  $\lambda = 0.04$   $\vdots$
  - 5. Try  $\lambda = 0.08$   $\vdots$
  - ⋮
  - 12. Try  $\lambda = 10$   $\rightarrow \theta^{(12)} \rightarrow J_{cv}(\theta^{(12)})$
- Pick (say)  $\theta^{(5)}$ . Test error:  $J_{test}(\theta^{(5)})$

Andrew

So, this gives me maybe 12 different models, and I'm trying to select one of them, corresponding to 12 different values of the regularization parameter lambda. And of course you can also go to values less than 0.01 or values larger than 10 but I've just truncated it here for convenience. Given each of these 12 models, what we can do is then the following.

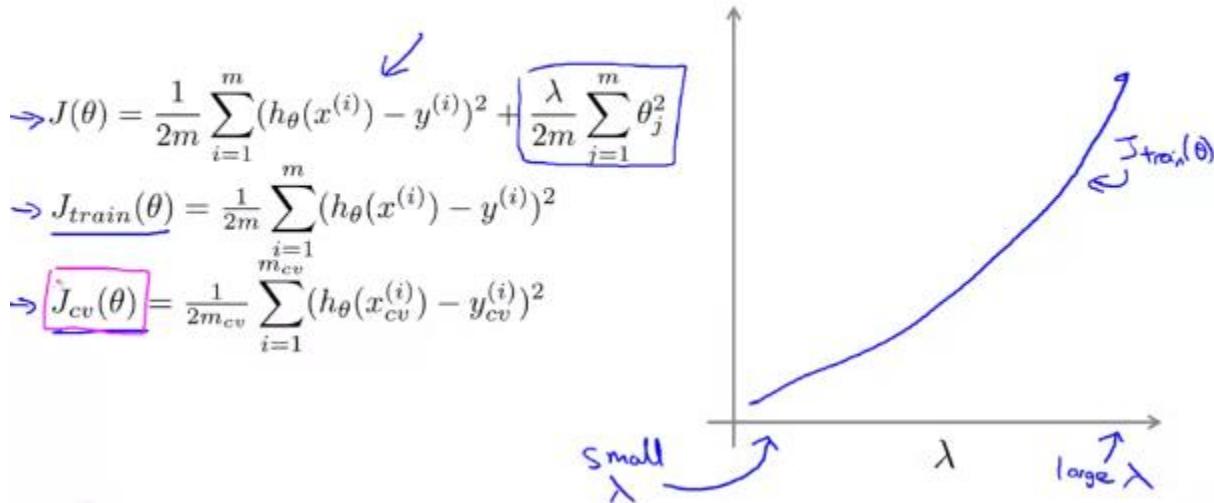
We can take this first model with lambda equals zero and minimize my cost function  $J(\theta)$  and this will give me some parameter vector  $\theta$ . And similar to the earlier video, let me just denote this as  $\theta^{(1)}$ . And then I can take my second model with lambda set to 0.01 and minimize my cost function, now using lambda equals 0.01 of course, to get some different parameter vector  $\theta$ . Let me denote that  $\theta^{(2)}$ . And for that I end up with  $\theta^{(3)}$ . So this is part for my third model. And so on until for my final model with lambda set to 10 or 10.24, I end up with this  $\theta^{(12)}$ .

Next, I can talk all of these hypotheses, all of these parameters and use my cross validation set to validate them. So I can look at my first model, my second model, fit to these different values of the regularization parameter, and evaluate them with my cross validation set, basically measure the average square error of each of these parameter vectors theta on my cross validation set.

And I would then pick whichever one of these 12 models gives me the lowest error on the cross validation set. And let's say, for the sake of this example, that I end up picking  $\theta^{(5)}$ , the 5th order polynomial, because that has the lowest cross validation error. Having done that, finally what I would do if I wanted to report each test set error, is to take the parameter  $\theta^{(5)}$  that I've selected, and look at how well it does on my test set. So once again, here is as if we've fit this parameter,  $\theta$ , to my cross-validation set, which is why I'm setting aside a separate test set that I'm going to use to get a better estimate of how well my parameter vector  $\theta$  will generalize to previously unseen examples.

So that's model selection applied to selecting the regularization parameter lambda.

### Bias/variance as a function of the regularization parameter $\lambda$



The last thing I'd like to do in this video is get a better understanding of how cross validation and training error vary as we vary the regularization parameter lambda. And so just a reminder, right, that was our original cost function  $J(\theta)$ . But for this purpose we're going to define training error without using a regularization parameter, and cross validation error without using the regularization parameter. And what I'd like to do is plot this of  $J_{train}(\theta)$ , and plot this  $J_{cv}(\theta)$ , meaning just how well does my hypothesis do on the training set and how well does my hypothesis do on cross validation set as I vary my regularization parameter lambda.

So as we saw earlier, if lambda is small then we're not using much regularization and we run a larger risk of over fitting, whereas if lambda is large that is if we were on the right part of this horizontal axis then, with a large value of lambda, we run a higher risk of having a bias problem, so if you plot  $J_{train}(\theta)$  and  $J_{cv}(\theta)$ , what you find is that, for small values of lambda you can fit the training set relatively well because you're not regularizing, so for small values of lambda the regularization term basically goes away, and you're just minimizing pretty much just square error. So when lambda is small, you end up with a small value for  $J_{train}(\theta)$ , whereas if lambda is large, then you have a high bias problem, and you might not fit your training set well, so you end up with the value up there.

So  $J_{train}(\theta)$  will tend to increase when lambda increases, because a large value of lambda corresponds to high bias where you might not even fit your trainings that well, whereas a small value of lambda corresponds to, if you can really fit a very high degree polynomial to your data, let's say.

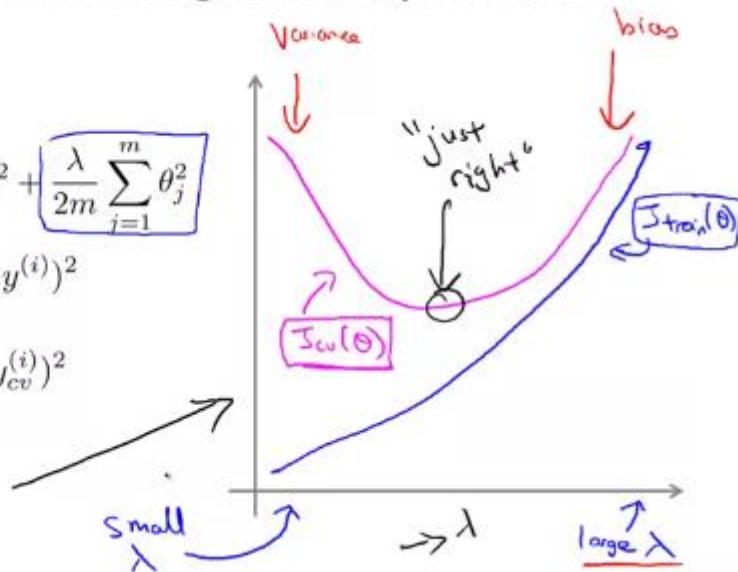
After the cost validation error we end up with a figure like this, where over here on the right, if we have a large value of lambda, we may end up under fitting, and so this is the bias regime.

## Bias/variance as a function of the regularization parameter $\lambda$

$$\Rightarrow J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2 + \boxed{\frac{\lambda}{2m} \sum_{j=1}^m \theta_j^2}$$

$$\Rightarrow \underline{J_{train}(\theta)} = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$$

$$\Rightarrow \boxed{J_{cv}(\theta)} = \frac{1}{2m_{cv}} \sum_{i=1}^{m_{cv}} (h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$$



Whereas, and so the cross validation error will be high. Let me just leave all of that to this  $J_{cv}(\theta)$  because so, with high bias, we won't be fitting, we won't be doing well on the cross validation set, whereas here on the left, this is the high variance regime, where if we have too small a value of lambda, then we may be over fitting the data. And so by over fitting the data, then the cross validation error will also be high. And so, this is what the cross validation error and what the training error may look like on a training set as we vary the regularization parameter lambda.

And so once again, it will often be some intermediate value of lambda that is just right or that works best In terms of having a small cross validation error or a small test theta.

And whereas the curves I've drawn here are somewhat cartoonish and somewhat idealized so on the real data set the curves you get may end up looking a little bit more messy and just a little bit more noisy then this. For some data sets you will really see these four sorts of trends and by looking at a plot of the hold-out cross validation error you can either manually, automatically try to select a point that minimizes the cross validation error and select the value of lambda corresponding to low cross validation error.

When I'm trying to pick the regularization parameter lambda for learning algorithm, often I find that plotting a figure like this one shown here helps me understand better what's going on and helps me verify that I am indeed picking a good value for the regularization parameter lambda.

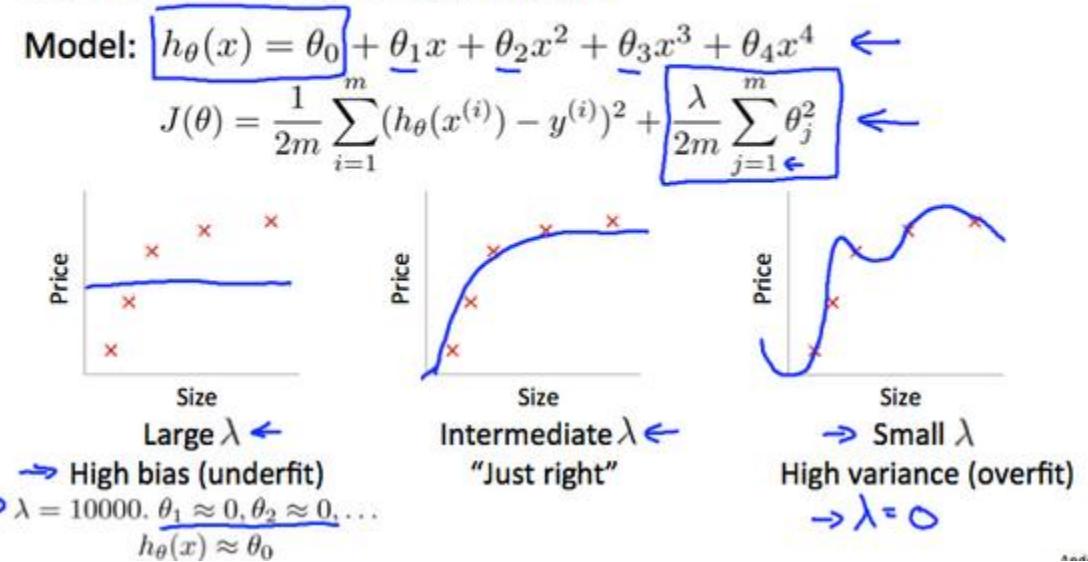
So hopefully that gives you more insight into regularization and its effects on the bias and variance of a learning algorithm.

By now you've seen bias and variance from a lot of different perspectives. And what we like to do in the next video is take all the insights we've gone through and build on them to put together a diagnostic that's called learning curves, which is a tool that I often use to try to diagnose if the learning algorithm may be suffering from a bias problem or a variance problem, or a little bit of both.

## Summary: Regularization and Bias/Variance

**Note:** [The regularization term below and through out the video should be  $\frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$  and NOT  $\frac{\lambda}{2m} \sum_{j=1}^m \theta_j^2$ ]

### Linear regression with regularization



In the figure above, we see that as  $\lambda$  increases, our fit becomes more rigid. On the other hand, as  $\lambda$  approaches 0, we tend to overfit the data. So how do we choose our parameter  $\lambda$  to get it 'just right'? In order to choose the model and the regularization term  $\lambda$ , we need to:

1. Create a list of lambdas (i.e.  $\lambda \in \{0, 0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64, 1.28, 2.56, 5.12, 10.24\}$ );
2. Create a set of models with different degrees or any other variants.
3. Iterate through the  $\lambda$  and for each  $\lambda$  go through all the models to learn some  $\theta$ .
4. Compute the cross validation error using the learned  $\theta$  (computed with  $\lambda$ ) on the  $J_{cv}(\theta)$  **without** regularization or  $\lambda = 0$ .
5. Select the best combo that produces the lowest error on the cross validation set.
6. Using the best combo  $\theta$  and  $\lambda$ , apply it on  $J_{test}(\theta)$  to see if it has a good generalization of the problem.

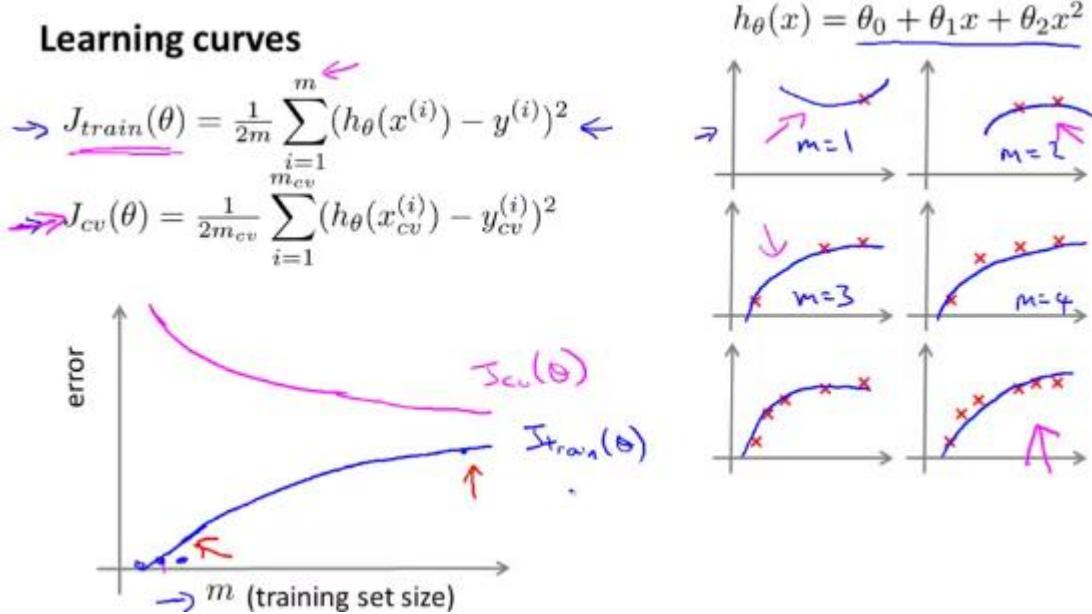
## Learning Curves

In this video, I'd like to tell you about learning curves.

Learning curves is often a very useful thing to plot. If either you wanted to sanity check that your algorithm is working correctly, or if you want to improve the performance of the algorithm. And

learning curves is a tool that I actually use very often to try to diagnose if a physical learning algorithm may be suffering from bias or a variance problem or a bit of both.

Here's what a learning curve is.



To plot a learning curve, what I usually do is plot  $J_{train}(\theta)$ , which is the, say, average squared error on my training set or  $J_{cv}(\theta)$ , which is the average squared error on my cross validation set. And I'm going to plot that as a function of  $m$ , that is as a function of the number of training examples I have. And so  $m$  is usually a constant like maybe I just have, you know, a 100 training examples but what I'm going to do is artificially reduce my training set size, so I deliberately limit myself to using only, say, 10 or 20 or 30 or 40 training examples and plot what the training error is and what the cross validation error is for these smaller training set sizes. So let's see what these plots may look like.

Suppose I have only one training example like that shown in this first example here and let's say I'm fitting a quadratic function. Well, I have only one training example. I'm going to be able to fit it perfectly right? You know, just fit the quadratic function. I'm going to have 0 error on the one training example. If I have two training examples. Well the quadratic function can also fit that very well. So, even if I am using regularization, I can probably fit this quite well. And if I am using no regularization, I'm going to fit this perfectly and if I have three training examples again, yeah, I can fit a quadratic function perfectly. So if  $m$  equals 1 or  $m$  equals 2 or  $m$  equals 3, my training error on my training set is going to be 0 assuming I'm not using regularization or it may slightly large than 0 if I'm using regularization. And by the way if I have a large training set and I'm artificially restricting the size of my training set in order to apply  $J_{train}(\theta)$ , here if I set  $m$  equals 3, say, and I train on only three examples, then, for this figure I am going to measure my training error only on the three examples that actually fit my data to and so even if I have say a 100 training examples but if I want to plot what my training error is when  $m$  equals 3, what I'm going to do is to measure the training error only on the three examples that I've actually fit to my hypothesis 2, and not on all the other examples that I have deliberately omitted from the training process.

So just to summarize what we've seen is that if the training set size is small then the training error is going to be small as well.

Because you know, we have a small training set is going to be very easy to fit your training set very well, maybe even perfectly. Now, say we have  $m$  equals 4 examples. Well then a quadratic function can no longer fit this data set perfectly and if I have  $m$  equals 5 then you know, maybe quadratic function will fit this data so so, then as my training set gets larger it becomes harder and harder to ensure that I can find the quadratic function that processes through all my examples perfectly. So in fact as the training set size grows what you find is that my average training error actually increases and so if you plot this figure what you find is that the training set error that is the average error on your hypothesis grows as  $m$  grows, and just to repeat the intuition is that when  $m$  is small when you have very few training examples it's pretty easy to fit every single one of your training examples perfectly and so your error is going to be small, whereas when  $m$  is larger then gets so much harder all the training examples perfectly and so your training set error becomes all the more larger.

Now, how about the cross validation error.

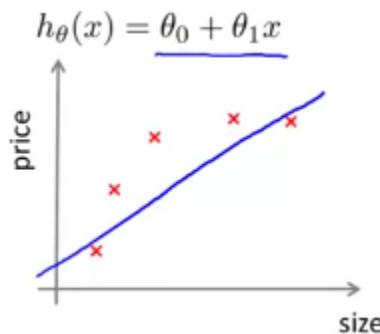
Well, the cross validation error is my error on this cross validation set that I haven't seen and so, you know, when I have a very small training set, I'm not going to generalize well, just not going to do well on that. So, right, this hypothesis here doesn't look like a good one, and it's only when I get a larger training set that, you know, I'm starting to get hypotheses that maybe fit the data somewhat better. So, your cross validation error and your test set error will tend to decrease as your training set size increases because the more data you have, the better you do at generalizing to new examples.

So, just the more data you have, the better the hypothesis you fit. So if you plot  $J_{\text{train}}(\theta)$ , and  $J_{\text{cv}}(\theta)$ , this is the sort of thing that you get.

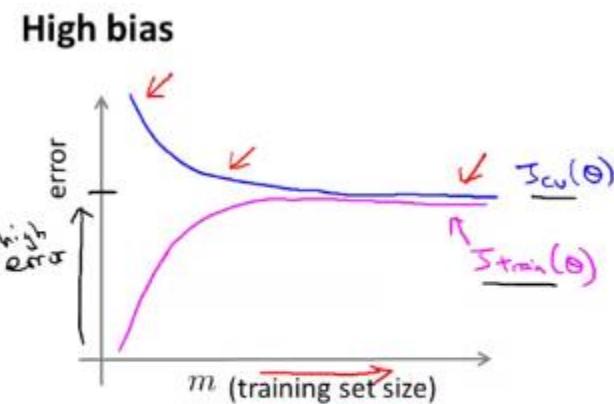
Now let's look at what the learning curves may look like if we have either high bias or high variance problems.

==

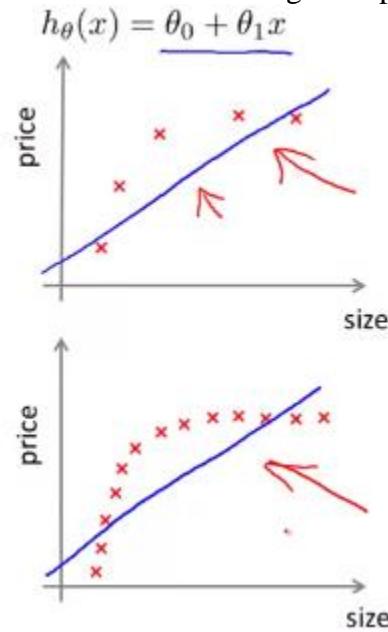
Suppose your hypothesis has high bias and to explain this I'm going to use a, set an example of fitting a straight line to data that, you know can't really be fit well by a straight line. So we end up with a hypotheses that maybe looks like that.



Now let's think what would happen if we were to increase the training set size. So if instead of five examples like what I've drawn there, imagine that we have a lot more training examples.



If a learning algorithm is suffering from high bias, getting more training data will not (by itself) help much.



Well what happens, if you fit a straight line to this. What you find is that, you end up with you know, pretty much the same straight line. I mean a straight line that just cannot fit this data and getting a ton more data, well the straight line isn't going to change that much. This is the best possible straight-line fit to this data, but the straight line just can't fit this data set that well. So, if you plot across validation error, this is what it will look like.

On the left, if you have already a minuscule training set size like you know, maybe just one training example and is not going to do well. But by the time you have reached a certain number of training examples, you have almost fit the best possible straight line, and even if you end up with a much larger training set size, a much larger value of  $m$ , you know, you're basically getting the same straight line, and so, the cross-validation error - let me label that - or test set error or plateau out, or flatten out pretty soon, once you reached beyond a certain number of training examples, unless you pretty much fit the best possible straight line.

And how about training error? Well, the training error will again be small. And what you find in the high bias case is that the training error will end up close to the cross validation error, because you have so few parameters and so much data, at least when  $m$  is large, the performance on the training set and the cross validation set will be very similar. And so, this is what your learning curves will look like, if you have an algorithm that has high bias.

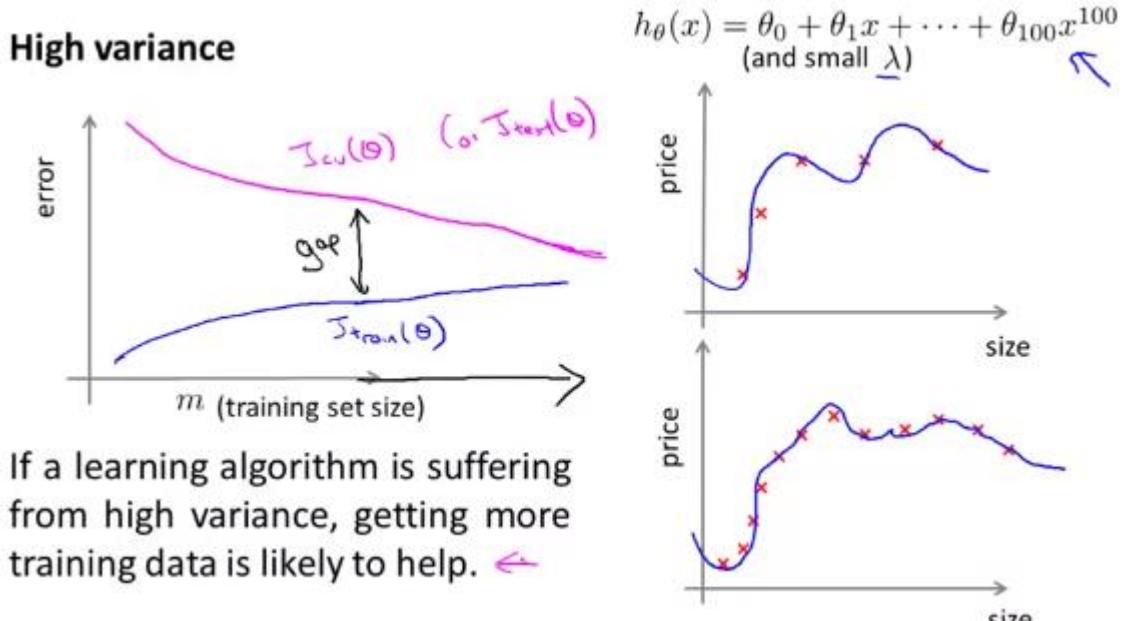
And finally, the problem with high bias is reflected in the fact that both the cross validation error and the training error are high, and so you end up with a relatively high value of both  $J_{cv}(\theta)$  and  $J_{train}(\theta)$ .

This also implies something very interesting, which is that, if a learning algorithm has high bias, as we get more and more training examples, that is, as we move to the right of this figure, we'll notice that the cross validation error isn't going down much, it's basically flattened out, and so if

learning algorithms is already suffering from high bias getting more training data by itself will actually not help that much and as a concrete example in the figure on the right here we had only five training examples, and we fit a certain straight line. And when we had a ton more training data, we still end up with roughly the same straight line. And so if the learning algorithm has high bias just giving a lot more training data that doesn't actually help you get a much lower cross validation error or test set error.

So knowing if your learning algorithm is suffering from high bias seems like a useful thing to know because this can prevent you from wasting a lot of time collecting more training data where it might just not end up being helpful.

Next let us look at the setting of a learning algorithm that may have high variance.



Let us just look at the training error. If you have very small training set like five training examples shown on the figure on the right and if we're fitting say a very high order polynomial, and I've written a hundredth degree polynomial which really no one uses, but just an illustration. And if we're using a fairly small value of lambda, maybe not zero, but a fairly small value of lambda, then we'll end up, you know, fitting this data very well that with a function that overfits this.

So, if the training set size is small, our training error, that is,  $J_{\text{train}}(\theta)$  will be small. And as this training set size increases a bit, you know, we may still be overfitting this data a little bit but it also becomes slightly harder to fit this data set perfectly, and so, as the training set size increases, we'll find that  $J_{\text{train}}(\theta)$  increases because it is just a little harder to fit the training set perfectly when we have more examples, but the training set error will still be pretty low.

Now, how about the cross validation error? Well, in high variance setting, a hypothesis is overfitting and so the cross validation error will remain high, even as we get you know, a moderate number of training examples and, so maybe, the cross validation error may look like that. And the indicative diagnostic that we have a high variance problem, is the fact that there's

this large gap between the training error and the cross validation error. And looking at this figure, if we think about adding more training data, that is, taking this figure and extrapolating to the right, we can kind of tell that, you know the two curves, the blue curve and the magenta curve, are converging to each other. And so, if we were to extrapolate this figure to the right, then it seems it likely that the training error will keep on going up and the cross-validation error would keep on going down. And the thing we really care about is the cross-validation error or the test set error, right? So in this sort of figure, we can tell that if we keep on adding training examples and extrapolate to the right, well our cross validation error will keep on coming down. And, so, in the high variance setting, getting more training data is, indeed, likely to help. And so again, this seems like a useful thing to know if your learning algorithm is suffering from a high variance problem, because that tells you, for example that it may be worth a while to see if you can go and get some more training data.

Now, on the previous slide and this slide, I've drawn fairly clean fairly idealized curves. If you plot these curves for an actual learning algorithm, sometimes you will actually see, you know, pretty much curves like what I've drawn here. Although, sometimes you see curves that are a little bit noisier and a little bit messier than this.

But plotting learning curves like these can often tell you, can often help you figure out if your learning algorithm is suffering from bias, or variance or even a little bit of both. So when I'm trying to improve the performance of a learning algorithm, one thing that I'll almost always do is plot these learning curves, and usually this will give you a better sense of whether there is a bias or variance problem.

And in the next video we'll see how this can help suggest specific actions to take, or to not take, in order to try to improve the performance of your learning algorithm.

## Summary: Learning Curves

Training an algorithm on a very few number of data points (such as 1, 2 or 3) will easily have 0 errors because we can always find a quadratic curve that touches exactly those number of points. Hence:

- As the training set gets larger, the error for a quadratic function increases.
- The error value will plateau out after a certain m, or training set size.

### Experiencing high bias:

**Low training set size:** causes  $J_{\text{train}}(\theta)$  to be low and  $J_{\text{CV}}(\theta)$  to be high.

**Large training set size:** causes both  $J_{\text{train}}(\theta)$  and  $J_{\text{CV}}(\theta)$  to be high with  $J_{\text{train}}(\theta) \approx J_{\text{CV}}(\theta)$ .

If a learning algorithm is suffering from **high bias**, getting more training data will not (**by itself**) help much.

**More on Bias vs. Variance**

Typical learning curve for high bias(at fixed model complexity):

**Experiencing high variance:**

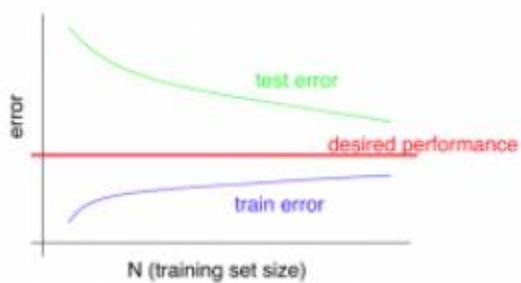
**Low training set size:**  $J_{\text{train}}(\Theta)$  will be low and  $J_{\text{CV}}(\Theta)$  will be high.

**Large training set size:**  $J_{\text{train}}(\Theta)$  increases with training set size and  $J_{\text{CV}}(\Theta)$  continues to decrease without leveling off. Also,  $J_{\text{train}}(\Theta) < J_{\text{CV}}(\Theta)$  but the difference between them remains significant.

If a learning algorithm is suffering from **high variance**, getting more training data is likely to help.

**More on Bias vs. Variance**

Typical learning curve for high variance(at fixed model complexity):

**Deciding What to Do Next Revisited**

We've talked about how to evaluate learning algorithms, talked about model selection, talked a lot about bias and variance.

So how does this help us figure out what are potentially fruitful, potentially not fruitful things to try to do to improve the performance of a learning algorithm. Let's go back to our original motivating example and go for the result.

So here is our earlier example of maybe having fit regularized linear regression and finding that it doesn't work as well as we're hoping. We said that we had this menu of options. So is there some way to figure out which of these might be fruitful options?

### **Debugging a learning algorithm:**

Suppose you have implemented regularized linear regression to predict housing prices. However, when you test your hypothesis in a new set of houses, you find that it makes unacceptably large errors in its prediction. What should you try next?

- Get more training examples → fixes high variance
- Try smaller sets of features → fixes high variance
- Try getting additional features → fixes high bias
- Try adding polynomial features ( $x_1^2, x_2^2, x_1x_2$ , etc) → fixes high bias.
- Try decreasing  $\lambda$  → fixes high bias
- Try increasing  $\lambda$  → fixes high variance

The first thing all of this was getting more training examples. And what this is good for, is this helps to fix high variance. And concretely, if you instead have a high bias problem and don't have any variance problem, then we saw in the previous video that getting more training examples, while maybe just isn't going to help much at all.

So the first option is useful only if you, say, plot the learning curves and figure out that you have at least a bit of a variance, meaning that the cross-validation error is, you know, quite a bit bigger than your training set error. How about trying a smaller set of features? Well, trying a smaller set of features, that's again something that fixes high variance.

And in other words, if you figure out, by looking at learning curves or something else that you used, that have a high bias problem; then for goodness sakes, don't waste your time trying to carefully select out a smaller set of features to use. Because if you have a high bias problem, using fewer features is not going to help.

Whereas in contrast, if you look at the learning curves or something else you figure out that you have a high variance problem, then, indeed trying to select out a smaller set of features that might indeed be a very good use of your time.

How about trying to get additional features, adding features, usually, not always, but usually we think of this as a solution for fixing high bias problems. So if you are adding extra features it's usually because your current hypothesis is too simple, and so we want to try to get additional features to make our hypothesis better able to fit the training set.

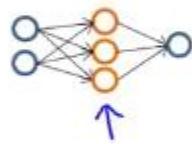
And similarly, adding polynomial features; this is another way of adding features and so there is another way to try to fix the high bias problem. And, if concretely, if your learning curves show you that you still have a high variance problem, then, you know, again this is maybe a less good use of your time.

And finally, decreasing and increasing lambda. These are quick and easy to try, I guess these are less likely to be a waste of, you know, many months of your life. But decreasing lambda, you already know fixes high bias. In case this isn't clear to you, you know, I do encourage you to pause the video and think through this that convince yourself that decreasing lambda helps fix high bias, whereas increasing lambda fixes high variance. And if you aren't sure why this is the case, do pause the video and make sure you can convince yourself that this is the case. Or take a look at the curves that we were plotting at the end of the previous video and try to make sure you understand why these are the case.

Finally, let us take everything we have learned and relate it back to neural networks and so, here is some practical advice for how I usually choose the architecture or the connectivity pattern of the neural networks I use.

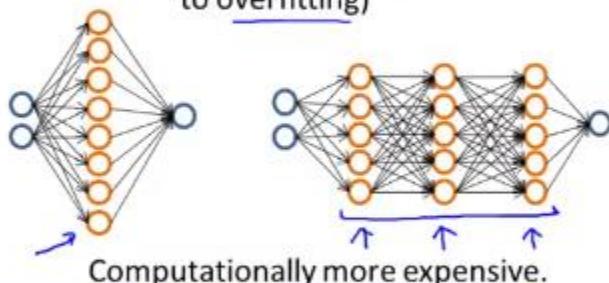
### Neural networks and overfitting

→ “Small” neural network  
(fewer parameters; more prone to underfitting)



Computationally cheaper

→ “Large” neural network  
(more parameters; more prone to overfitting)



Computationally more expensive.

Use regularization ( $\lambda$ ) to address overfitting.

$$\mathcal{J}_{\text{reg}}(\theta)$$

So, if you are fitting a neural network, one option would be to fit, say, a pretty small neural network with you know, relatively few hidden units, maybe just one hidden unit. If you're fitting a neural network, one option would be to fit a relatively small neural network with, say, relatively few, maybe only one hidden layer and maybe only a relatively few number of hidden units.

So, a network like this might have relatively few parameters and be more prone to underfitting. The main advantage of these small neural networks is that they are computationally cheaper.

An alternative would be to fit a, maybe relatively large neural network with either more hidden units -- there's a lot of hidden in one layer -- or with more hidden layers. And so these neural networks tend to have more parameters and therefore be more prone to overfitting. One disadvantage, often not a major one but something to think about, is that if you have a large number of neurons in your network, then it can be more computationally expensive. Although within reason, this is often hopefully not a huge problem. The main potential problem of these much larger neural networks is that it could be more prone to overfitting and it turns out that if you're applying neural network very often using a large neural network often it's actually the larger, the better but if it's overfitting, you can then use regularization to address overfitting, And

usually using a larger neural network by using regularization to address overfitting that's often more effective than using a smaller neural network. And the main possible disadvantage is that it can be more computationally expensive.

And finally, one of the other decisions is, say, the number of hidden layers you want to have, right? So, do you want one hidden layer or do you want three hidden layers, as we've shown here, or do you want two hidden layers?

And usually, as I think I said in the previous video, using a single hidden layer is a reasonable default, but if you want to choose the number of hidden layers, one other thing you can try is find yourself a training cross-validation, and test set split and try training neural networks with one hidden layer or two hidden layers or three hidden layers and see which of those neural networks performs best on the cross-validation sets. You take your three neural networks with one, two and three hidden layers, and compute the cross validation error at  $J_{cv}(\theta)$  and all of them and use that to select which of these is you think the best neural network.

So, that's it for bias and variance and ways like learning curves, we tried to diagnose these problems. As far as what you think is implied, for one might be truthful or not truthful things to try to improve the performance of a learning algorithm. If you understood the contents of the last few videos and if you apply them you actually be much more effective already and getting learning algorithms to work on problems and even a large fraction, maybe the majority of practitioners of machine learning here in Silicon Valley today doing these things as their full-time jobs.

So I hope that these pieces of advice and by experience in diagnostics will help you to much more effectively and powerfully apply learning algorithms and get them to work very well.

## Summary: Deciding What to Do Next Revised

Our decision process can be broken down as follows:

- **Getting more training examples:** Fixes high variance
- **Trying smaller sets of features:** Fixes high variance
- **Adding features:** Fixes high bias
- **Adding polynomial features:** Fixes high bias
- **Decreasing  $\lambda$ :** Fixes high bias
- **Increasing  $\lambda$ :** Fixes high variance.

## Diagnosing Neural Networks

- A neural network with fewer parameters is **prone to underfitting**. It is also **computationally cheaper**.

- A large neural network with more parameters is **prone to overfitting**. It is also **computationally expensive**. In this case you can use regularization (increase  $\lambda$ ) to address the overfitting.

Using a single hidden layer is a good starting default. You can train your neural network on a number of hidden layers using your cross validation set. You can then select the one that performs best.

### **Model Complexity Effects:**

- Lower-order polynomials (low model complexity) have high bias and low variance. In this case, the model fits poorly consistently.
- Higher-order polynomials (high model complexity) fit the training data extremely well and the test data extremely poorly. These have low bias on the training data, but very high variance.
- In reality, we would want to choose a model somewhere in between, that can generalize well but also fits the data reasonably well.

## **Prioritizing What to Work On**

In the next few videos I'd like to talk about machine learning system design. These videos will touch on the main issues that you may face when designing a complex machine learning system, and will actually try to give advice on how to strategize putting together a complex machine learning system.

In case this next set of videos seems a little disjointed that's because these videos will touch on a range of the different issues that you may come across when designing complex learning systems. And even though the next set of videos may seem somewhat less mathematical, I think that this material may turn out to be very useful, and potentially huge time savers when you're building big machine learning systems.

Concretely, I'd like to begin with the issue of prioritizing how to spend your time on what to work on, and I'll begin with an example on spam classification.

Let's say you want to build a spam classifier. Here are a couple of examples of obvious spam and non-spam emails. If the one on the left tried to sell things. And notice how spammers will deliberately misspell words, like Vincent with a 1 there, and mortgages. And on the right as maybe an obvious example of non-spam email, actually email from my younger brother.

## Building a spam classifier

From: cheapsales@buystufffromme.com  
 To: ang@cs.stanford.edu  
 Subject: Buy now!

Deal of the week! Buy now!  
Rolex w4tchs - \$100  
Med1cine (any kind) - \$50  
 Also low cost M0rgages  
 available.

Spam (1)

From: Alfred Ng  
 To: ang@cs.stanford.edu  
 Subject: Christmas dates?

Hey Andrew,  
 Was talking to Mom about plans  
 for Xmas. When do you get off  
 work. Meet Dec 22?  
 Alf

Non-spam (0)

Let's say we have a labeled training set of some number of spam emails and some non-spam emails denoted with labels  $y$  equals 1 or 0, how do we build a classifier using supervised learning to distinguish between spam and non-spam?

### Building a spam classifier

Supervised learning.  $x = \text{features of email}$ .  $y = \text{spam (1) or not spam (0)}$ .

Features  $x$ : Choose 100 words indicative of spam/not spam.

E.g. deal, buy, discount, andrew, now, ...

$$x = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \begin{matrix} \text{andrew} \\ \text{buy} \\ \text{deal} \\ \text{discount} \\ \vdots \\ \text{now} \end{matrix} \quad x \in \mathbb{R}^{100}$$

$$x_j = \begin{cases} 1 & \text{if word } j \text{ appears} \\ 0 & \text{otherwise.} \end{cases}$$

From: cheapsales@buystufffromme.com  
 To: ang@cs.stanford.edu  
 Subject: Buy now!

Deal of the week! Buy now!

Note: In practice, take most frequently occurring  $n$  words (10,000 to 50,000) in training set, rather than manually pick 100 words.

In order to apply supervised learning, the first decision we must make is how do we want to represent  $x$ , that is the features of the email. Given the features  $x$  and the labels  $y$  in our training set, we can then train a classifier, by example using logistic regression.

Here's one way to choose a set of features for our emails. We could come up with, say, a list of maybe a hundred words that we think are indicative of whether e-mail is spam or non-spam, for example, if a piece of e-mail contains the word 'deal' maybe it's more likely to be spam if it contains the word 'buy' maybe more likely to be spam, a word like 'discount' is more likely to be spam, whereas if a piece of email contains my name, Andrew, maybe that means the person actually knows who I am and that might mean it's less likely to be spam. And maybe for some reason I think the word "now" may be indicative of non-spam because I get a lot of urgent

emails, and so on, and maybe we choose a hundred words or so. Given a piece of email, we can then take this piece of email and encode it into a feature vector as follows.

I'm going to take my list of a hundred words and sort them in alphabetical order say. It doesn't have to be sorted. But, you know, here's a, here's my list of words, just count and so on, until eventually I'll get down to now, and so on and given a piece of e-mail like that shown on the right, I'm going to check and see whether or not each of these words appears in the e-mail and then I'm going to define a feature vector  $x$  where in this piece of an email on the right, my name doesn't appear so I'm gonna put a zero there. The word "buy" does appear, so I'm gonna put a one there and I'm just gonna put one's or zeroes. I'm gonna put a one even though the word "buy" occurs twice. I'm not gonna recount how many times the word occurs. The word "deal" appears, I put a one there. The word "discount" doesn't appear, at least not in this this little short email, and so on. The word "now" does appear and so on. So I put ones and zeroes in this feature vector depending on whether or not a particular word appears. And in this example my feature vector would have to mention one hundred, if I have a hundred, if I chose a hundred words to use for this representation; and each of my features  $X_j$  will basically be 1 if you have a particular word that, we'll call this word  $j$ , appears in the email and  $X_j$  would be zero otherwise.

Okay. So that gives me a representation, a feature representation of a piece of email. By the way, even though I've described this process as manually picking a hundred words, in practice what's most commonly done is to look through a training set, and in the training set depict the most frequently occurring  $n$  words where  $n$  is usually between ten thousand and fifty thousand, and use those as your features.

So rather than manually picking a hundred words, here you look through the training examples and pick the most frequently occurring words like ten thousand to fifty thousand words, and those form the features that you are going to use to represent your email for spam classification.

### **Building a spam classifier**

#### **How to spend your time to make it have low error?**

- Collect lots of data
  - E.g. "honeypot" project.
- Develop sophisticated features based on email routing information (from email header).
- Develop sophisticated features for message body, e.g. should "discount" and "discounts" be treated as the same word? How about "deal" and "Dealer"? Features about punctuation?
- Develop sophisticated algorithm to detect misspellings (e.g. m0rtgage, med1cine, w4tches.)

Now, if you're building a spam classifier one question that you may face is, what's the best use of your time in order to make your spam classifier have higher accuracy, you have lower error. One natural inclination is going to collect lots of data. Right?

And in fact there's this tendency to think that, well the more data we have the better the algorithm will do. And in fact, in the email spam domain, there are actually pretty serious

projects called Honey Pot Projects, which create fake email addresses and try to get these fake email addresses into the hands of spammers and use that to try to collect tons of spam email, and therefore you know, get a lot of spam data to train learning algorithms. But we've already seen in the previous sets of videos that getting lots of data will often help, but not all the time.

But for most machine learning problems, there are a lot of other things you could usually imagine doing to improve performance. For spam, one thing you might think of is to develop more sophisticated features on the email, maybe based on the email routing information. And this would be information contained in the email header. So, when spammers send email, very often they will try to obscure the origins of the email, and maybe use fake email headers. Or send email through very unusual sets of computer service. Through very unusual routes, in order to get the spam to you. And some of this information will be reflected in the email header. And so one can imagine, looking at the email headers and trying to develop more sophisticated features to capture this sort of email routing information to identify if something is spam.

Something else you might consider doing is to look at the email message body, that is the email text, and try to develop more sophisticated features. For example, should the word 'discount' and the word 'discounts' be treated as the same words or should we have treat the words 'deal' and 'dealer' as the same word? Maybe even though one is lower case and one in capitalized in this example. Or do we want more complex features about punctuation because maybe spam is using exclamation marks a lot more. I don't know.

And along the same lines, maybe we also want to develop more sophisticated algorithms to detect and maybe to correct to deliberate misspellings, like mortgage, medicine, watches. Because spammers actually do this, because if you have watches with a 4 in there then well, with the simple technique that we talked about just now, the spam classifier might not equate this as the same thing as the word "watches," and so it may have a harder time realizing that something is spam with these deliberate misspellings. And this is why spammers do it.

While working on a machine learning problem, very often you can brainstorm lists of different things to try, like these. By the way, I've actually worked on the spam problem myself for a while. And I actually spent quite some time on it. And even though I kind of understand the spam problem, I actually know a bit about it, I would actually have a very hard time telling you of these four options which is the best use of your time.

So what happens, frankly what happens far too often is that a research group or product group will randomly fixate on one of these options. And sometimes that turns out not to be the most fruitful way to spend your time depending, you know, on which of these options someone ends up randomly fixating on.

By the way, in fact, if you even get to the stage where you brainstorm a list of different options to try, you're probably already ahead of the curve. Sadly, what most people do is instead of trying to list out the options of things you might try, what far too many people do is wake up one morning and, for some reason, just, you know, have a weird gut feeling that, "Oh let's have a huge honeypot project to go and collect tons more data" and for whatever strange reason just sort of wake up one morning and randomly fixate on one thing and just work on that for six months.

But I think we can do better. And in particular what I'd like to do in the next video is tell you about the concept of error analysis and talk about the way where you can try to have a more systematic way to choose amongst the options of the many different things you might work, and therefore be more likely to select what is actually a good way to spend your time, you know for the next few weeks, or next few days or the next few months.

## Summary: Prioritizing What to Work On

### System Design Example:

Given a data set of emails, we could construct a vector for each email. Each entry in this vector represents a word. The vector normally contains 10,000 to 50,000 entries gathered by finding the most frequently used words in our data set. If a word is to be found in the email, we would assign its respective entry a 1, else if it is not found, that entry would be a 0. Once we have all our  $x$  vectors ready, we train our algorithm and finally, we could use it to classify if an email is a spam or not.

### Building a spam classifier

Supervised learning.  $x = \text{features of email}$ .  $y = \text{spam (1) or not spam (0)}$ .

Features  $x$ : Choose 100 words indicative of spam/not spam.

E.g. deal, buy, discount, andrew, now, ...

$$x = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \begin{matrix} \text{andrew} \\ \text{buy} \\ \text{deal} \\ \text{discount} \\ \vdots \\ \text{now} \end{matrix} \quad x \in \mathbb{R}^{100}$$

$$x_j = \begin{cases} 1 & \text{if word } j \text{ appears in email} \\ 0 & \text{otherwise.} \end{cases}$$

From: cheapsales@buystufffromme.com  
To: ang@cs.stanford.edu  
Subject: Buy now!

Deal of the week! Buy now!

So how could you spend your time to improve the accuracy of this classifier?

- Collect lots of data (for example "honeypot" project but doesn't always work)
- Develop sophisticated features (for example: using email header data in spam emails)
- Develop algorithms to process your input in different ways (recognizing misspellings in spam).

It is difficult to tell which of the options will be most helpful.

## Error Analysis

In the last video I talked about how, when faced with a machine learning problem, there are often lots of different ideas for how to improve the algorithm.

In this video, let's talk about the concepts of error analysis, which will hopefully give you a way to more systematically make some of these decisions.

If you're starting work on a machine learning problem, or building a machine learning application. It's often considered very good practice to start, not by building a very complicated system with lots of complex features and so on. But to instead start by building a very simple algorithm that you can implement quickly.

### **Recommended approach**

- - Start with a simple algorithm that you can implement quickly.  
Implement it and test it on your cross-validation data.
- - Plot learning curves to decide if more data, more features, etc. are likely to help.
- - Error analysis: Manually examine the examples (in cross validation set) that your algorithm made errors on. See if you spot any systematic trend in what type of examples it is making errors on.

And when I start with a learning problem what I usually do is spend at most one day, like literally at most 24 hours to try to get something really quick and dirty. Frankly not at all sophisticated system but get something really quick and dirty running, and implement it and then test it on my cross-validation data.

Once you've done that you can then plot learning curves, this is what we talked about in the previous set of videos. But plot learning curves of the training and test errors to try to figure out if you're learning algorithm maybe suffering from high bias or high variance, or something else. And use that to try to decide if having more data, more features, and so on are likely to help. And the reason that this is a good approach is often, when you're just starting out on a learning problem, there's really no way to tell in advance. Whether you need more complex features, or whether you need more data, or something else. And it's just very hard to tell in advance, that is, in the absence of evidence, in the absence of seeing a learning curve. It's just incredibly difficult to figure out where you should be spending your time.

And it's often by implementing even a very, very quick and dirty implementation. And by plotting learning curves, that helps you make these decisions. So if you like you can to think of this as a way of avoiding what's sometimes called premature optimization in computer programming.

And this idea that says we should let evidence guide our decisions on where to spend our time rather than use gut feeling, which is often wrong.

In addition to plotting learning curves, one other thing that's often very useful to do is what's called error analysis. And what I mean by that is that when building say a spam classifier. I will often look at my cross validation set and manually look at the emails that my algorithm is making errors on. So look at the spam e-mails and non-spam e-mails that the algorithm is misclassifying and see if you can spot any systematic patterns in what type of examples it is misclassifying. And often, by doing that, this is the process that will inspire you to design new features. Or they'll tell you what are the current things or current shortcomings of the system. And give you the inspiration you need to come up with improvements to it.

Concretely, here's a specific example.

### Error Analysis

$m_{CV} = 500$  examples in cross validation set

Algorithm misclassifies 100 emails.

Manually examine the 100 errors, and categorize them based on:

- (i) What type of email it is *pharma, replica, steal passwords, ...*
- (ii) What cues (features) you think would have helped the algorithm classify them correctly.

Pharma: 12

→ Deliberate misspellings: 5

Replica/fake: 4

(mOrgage, med1cine, etc.)

Steal passwords: 53

→ Unusual email routing: 16

Other: 31

→ Unusual (spamming) punctuation: 32

Let's say you've built a spam classifier and you have 500 examples in your cross validation set. And let's say in this example that the algorithm has a very high error rate. And this classifies 100 of these cross validation examples. So what I do is manually examine these 100 errors and manually categorize them. Based on things like what type of email it is, what cues or what features you think might have helped the algorithm classify them correctly. So, specifically, by what type of email it is, if I look through these 100 errors, I might find that maybe the most common types of spam emails in these classifies are maybe emails on pharma or pharmacies, trying to sell drugs. Maybe emails that are trying to sell replicas such as fake watches, fake random things, maybe some emails trying to steal passwords, these are also called phishing emails, that's another big category of emails, and maybe other categories.

So in terms of classifying what type of email it is, I would actually go through and count up my hundred emails, maybe I find that 12 of them is label emails, or pharma emails, and maybe 4 of them are emails trying to sell replicas, that sell fake watches or something. And maybe I find that 53 of them are these what's called phishing emails, basically emails trying to persuade you to give them your password. And 31 emails are other types of emails. And it's by counting up the number of emails in these different categories that you might discover, for example that the algorithm is doing really, particularly poorly on emails trying to steal passwords. And that may suggest that it might be worth your effort to look more carefully at that type of email and see if you can come up with better features to categorize them correctly.

And, also what I might do is look at what cues or what additional features might have helped the algorithm classify the emails. So let's say that some of our hypotheses about things or features that might help us classify emails better are “trying to detect deliberate misspellings” versus “unusual email routing” versus “unusual spamming punctuation”. Such as if people use a lot of exclamation marks.

And once again I would manually go through and let's say I find five cases of this and 16 of this and 32 of this and a bunch of other types of emails as well. And if this is what you get on your cross validation set, then it really tells you that maybe deliberate spellings is a sufficiently rare phenomenon that maybe it's not worth all the time trying to write algorithms that detect that. But if you find that a lot of spammers are using, you know, unusual punctuation, then maybe that's a strong sign that it might actually be worth your while to spend the time to develop more sophisticated features based on the punctuation.

So this sort of error analysis, which is really the process of manually examining the mistakes that the algorithm makes, can often help guide you to the most fruitful avenues to pursue. And this also explains why I often recommend implementing a quick and dirty implementation of an algorithm.

### **The importance of numerical evaluation**

Should discount/discounts/discounted/discounting be treated as the same word?

Can use “stemming” software (E.g. “Porter stemmer”) universe/university.

Error analysis may not be helpful for deciding if this is likely to improve performance. Only solution is to try it and see if it works.

Need numerical evaluation (e.g., cross validation error) of algorithm's performance with and without stemming.

Without stemming: 5% error With stemming: 3% error

Distinguish upper vs. lower case (Mom/mom): 3.2%

What we really want to do is figure out what are the most difficult examples for an algorithm to classify. And very often for different algorithms, for different learning algorithms they'll often find similar categories of examples difficult. And by having a quick and dirty implementation, that's often a quick way to let you identify some errors and quickly identify what are the hard examples. So that you can focus your effort on those.

Lastly, when developing learning algorithms, one other useful tip is to make sure that you have a way, you have a numerical evaluation of your learning algorithm. And what I mean by that is that you if you're developing a learning algorithm, it's often incredibly helpful. If you have a way of evaluating your learning algorithm that just gives you back a single real number, maybe accuracy, maybe error. But the single real number that tells you how well your learning algorithm is doing. I'll talk more about this specific concept in later videos, but here's a specific example.

Let's say we're trying to decide whether or not we should treat words like discount, discounts, discounted, discounting as the same word. So you know maybe one way to do that is to just look at the first few characters in the word like, you know. If you just look at the first few characters of a word, then you figure out that maybe all of these words are, have roughly similar meanings.

In natural language processing, the way that this is done is actually using a type of software called stemming software. And if you ever want to do this yourself, search on a web-search engine for the Porter Stemmer, and that would be one reasonable piece of software for doing this sort of stemming, which will let you treat all these words, discount, discounts, and so on, as the same word. But using a stemming software that basically looks at the first few alphabets of a word, more or less, it can help, but it can hurt. And it can hurt because for example, the software may mistake the words universe and university as being the same thing. Because, you know, these two words start off with very similar, the same alphabets. So if you're trying to decide whether or not to use stemming software for a spam cross classifier, it's not always easy to tell. And in particular, error analysis may not actually be helpful for deciding if this sort of stemming idea is a good idea.

Instead, the best way to figure out if using stemming software is good to help your classifier is if you have a way to very quickly just try it and see if it works. And in order to do this, having a way to numerically evaluate your algorithm is going to be very helpful.

Concretely, maybe the most natural thing to do is to look at the cross validation error of the algorithm's performance with and without stemming. So, if you run your algorithm without stemming and end up with 5 percent classification error. And you rerun it and you end up with 3 percent classification error, then this decrease in error very quickly allows you to decide that it looks like using stemming is a good idea.

For this particular problem, there's a very natural, single, real number evaluation metric, namely the cross validation error. We'll see later examples where coming up with this sort of single, real number evaluation metric will need a little bit more work. But as we'll see in a later video, doing so would also then let you make these decisions much more quickly of say, whether or not to use stemming.

And, just as one more quick example, let's say that you're also trying to decide whether or not to distinguish between upper versus lower case.

So, you know, as the word, mom, where upper case M versus lower case m, should that be treated as the same word or as different words? Should this be treated as the same feature, or as different features? And so, once again, because we have a way to evaluate our algorithm. If you try this down here, if I stopped distinguishing upper and lower case, maybe I end up with 3.2 percent error. And I find that therefore, this does worse than if I use only stemming. So, this lets me very quickly decide to go ahead and to distinguish or to not distinguish between upper and lowercase.

So when you're developing a learning algorithm, very often you'll be trying out lots of new ideas and lots of new versions of your learning algorithm. If every time you try out a new idea, if you

end up manually examining a bunch of examples again to see if it got better or worse, that's gonna make it really hard to make decisions on. Do you use stemming or not? Do you distinguish upper and lower case or not? But by having a single real number evaluation metric, you can then just look and see, oh, did the arrow go up or did it go down? And you can use that to much more rapidly try out new ideas and almost right away tell if your new idea has improved or worsened the performance of the learning algorithm. And this will let you often make much faster progress.

So the recommended, strongly recommended way to do error analysis is on the cross validation set rather than the test set. But, you know, there are people that will do this on the test set, even though that's definitely a less mathematically appropriate, certainly a less recommended way thing to do than to do error analysis on your cross validation set.

Set to wrap up this video, when starting on a new machine learning problem, what I almost always recommend is to implement a quick and dirty implementation of your learning algorithm.

And I've almost never seen anyone spend too little time on this quick and dirty implementation. I've pretty much only ever seen people spend much too much time building their first, supposedly, quick and dirty implementation. So really, don't worry about it being too quick, or don't worry about it being too dirty. But really, implement something as quickly as you can. And once you have the initial implementation, this is then a powerful tool for deciding where to spend your time next. Because first you can look at the errors it makes, and do this sort of error analysis to see what other mistakes it makes, and use that to inspire further development. And second, assuming your quick and dirty implementation incorporated a single real number evaluation metric this can then be a vehicle for you to try out different ideas and quickly see if the different ideas you're trying out are improving the performance of your algorithm.

And therefore let you, maybe much more quickly make decisions about what things to fold in and what things to incorporate into your learning algorithm.

## Summary: Error Analysis

The recommended approach to solving machine learning problems is to:

- Start with a simple algorithm, implement it quickly, and test it early on your cross validation data.
- Plot learning curves to decide if more data, more features, etc. are likely to help.
- Manually examine the errors on examples in the cross validation set and try to spot a trend where most of the errors were made.

For example, assume that we have 500 emails and our algorithm misclassifies 100 of them. We could manually analyze the 100 emails and categorize them based on what type of emails they are. We could then try to come up with new cues and features that would help us classify these 100 emails correctly. Hence, if most of our misclassified emails are those which try to steal passwords, then we could find some features that are particular to those emails and add them to

our model. We could also see how classifying each word according to its root changes our error rate:

### The importance of numerical evaluation

Should discount/discounts/discounted/discounting be treated as the same word?

Can use “stemming” software (E.g. “Porter stemmer”) universe/university.

Error analysis may not be helpful for deciding if this is likely to improve performance. Only solution is to try it and see if it works.

Need numerical evaluation (e.g., cross validation error) of algorithm’s performance with and without stemming.

Without stemming: 5% error With stemming: 3% error

Distinguish upper vs. lower case (Mom/mom): 3.2%

It is very important to get error results as a single, numerical value. Otherwise it is difficult to assess your algorithm’s performance. For example if we use stemming, which is the process of treating the same word with different forms (fail/failing/failed) as one word (fail), and get a 3% error rate instead of 5%, then we should definitely add it to our model. However, if we try to distinguish between upper case and lower case letters and end up getting a 3.2% error rate instead of 3%, then we should avoid using this new feature. Hence, we should try new things, get a numerical value for our error rate, and based on our result decide whether we want to keep the new feature or not.

## Error Metrics for Skewed Classes

In the previous video, I talked about error analysis and the importance of having error metrics that is of having a single real number evaluation metric for your learning algorithm to tell how well it’s doing.

In the context of evaluation and of error metrics, there is one important case, where it’s particularly tricky to come up with an appropriate error metric, or evaluation metric, for your learning algorithm. That case is the case of what’s called skewed classes.

Let me tell you what that means.

## Cancer classification example

Train logistic regression model  $h_\theta(x)$ . ( $y = 1$  if cancer,  $y = 0$  otherwise)

Find that you got 1% error on test set.  
(99% correct diagnoses)

Only 0.50% of patients have cancer.

*skewed classes.*

```
function y = predictCancer(x)
    → y = 0; % ignore x!
    return
```

0.5% error

→ 99.2% accy (0.8% error)  
→ 99.5% accy (0.5% error)

Consider the problem of cancer classification, where we have features of medical patients and we want to decide whether or not they have cancer. So this is like the malignant versus benign tumor classification example that we had earlier. So let's say  $y$  equals 1 if the patient has cancer and  $y$  equals 0 if they do not. We have trained a logistic regression classifier and let's say we test our classifier on a test set and find that we get 1 percent error. So, we're making 99% correct diagnosis. Seems like a really impressive result, right. We're correct 99% percent of the time.

But now, let's say we find out that only 0.5 percent of patients in our training and test sets actually have cancer. So only half a percent of the patients that come through our screening process have cancer. In this case, the 1% error no longer looks so impressive. And in particular, here's a piece of code, here's actually a piece of non learning code that takes this input of features  $x$  and it ignores it. It just sets  $y$  equals 0 and always predicts, you know, nobody has cancer and this algorithm would actually get 0.5 percent error. So this is even better than the 1% error that we were getting just now and this is a non learning algorithm that you know, it is just predicting  $y$  equals 0 all the time. So this setting of when the ratio of positive to negative examples is very close to one of two extremes, where, in this case, the number of positive examples is much, much smaller than the number of negative examples because  $y$  equals one so rarely, this is what we call the case of skewed classes.

We just have a lot more of examples from one class than from the other class. And by just predicting  $y$  equals 0 all the time, or maybe by predicting  $y$  equals 1 all the time, an algorithm can do pretty well. So the problem with using classification error or classification accuracy as our evaluation metric is the following.

Let's say you have one learning algorithm that's getting 99.2% accuracy. So, that's a 0.8% error. Let's say you make a change to your algorithm and you now are getting 99.5% accuracy. That is 0.5% error. So, is this an improvement to the algorithm or not? One of the nice things about having a single real number evaluation metric is this helps us to quickly decide if we just need a change to algorithm or not. By going from 99.2% accuracy to 99.5% accuracy, you know, did we just do something useful or did we just replace our code with something that just predicts  $y$  equals zero more often?

So, if you have very skewed classes it becomes much harder to use just classification accuracy, because you can get very high classification accuracies or very low errors, and it's not always clear if doing so is really improving the quality of your classifier because predicting  $y$  equals 0 all the time doesn't seem like a particularly good classifier. But just predicting  $y$  equals 0 more often can bring your error down to, you know, maybe as low as 0.5%.

When we're faced with such skewed classes therefore we would want to come up with a different error metric or a different evaluation metric. One such evaluation metric are what's called precision/recall.

### Precision/Recall

		$y = 1$ in presence of rare class that we want to detect	
		Actual class	
		1	0
Predicted 1 class	1	True positive	False positive
	0	False negative	True negative

$\rightarrow$  Precision  
(Of all patients where we predicted  $y = 1$ , what fraction actually has cancer?)

$$\frac{\text{True positives}}{\#\text{predicted positive}} = \frac{\text{True positive}}{\text{True pos} + \text{False pos}}$$

$\rightarrow$  Recall  
(Of all patients that actually have cancer, what fraction did we correctly detect as having cancer?)

$$\frac{\text{True positives}}{\#\text{actual positive}} = \frac{\text{True positives}}{\text{True pos} + \text{False neg}}$$

$y = 0$   
 $\text{recall} = 0$

Let me explain what that is. Let's say we are evaluating a classifier on a test set. For the examples in the test set the actual class of that example in the test set is going to be either one or zero, right, if there is a binary classification problem. And what our learning algorithm will do is it will, you know, predict some value for the class and our learning algorithm will predict a value for each example in my test set and the predicted value will also be either one or zero.

So let me draw a two by two table as follows, depending on a full of these entries depending on what was the actual class and what was the predicted class. If we have an example where the actual class is one and the predicted class is one then that's called an example that's a true positive, meaning our algorithm predicted that it's positive and in reality the example is positive. If our learning algorithm predicted that something is negative, class zero, and the actual class is also class zero then that's what's called a true negative. We predicted zero and it actually is zero. To find the other two boxes, if our learning algorithm predicts that the class is one but the actual class is zero, then that's called a false positive. So that means our algorithm for the patient has cancer but in reality the patient has not. And finally, the last box is a zero. That's called a false negative because our algorithm predicted zero, but the actual class was one. And so, we have this little two by two table based on what was the actual class and what was the predicted class.

So here's a different way of evaluating the performance of our algorithm. We're going to compute two numbers. The first is called precision - and what that says is, of all the patients

where we've predicted that they have cancer, what fraction of them actually have cancer? So let me write this down, the precision of a classifier is the number of true positives divided by the number that we predicted as positive, right? So of all the patients that we went to those patients and we told them, "We think you have cancer." Of all those patients, what fraction of them actually have cancer? So that's called precision.

And another way to write this would be true positives and then in the denominator is the number of predicted positives, and so that would be the sum of the, you know, entries in this first row of the table. So it would be true positives divided by true positives, I'm going to abbreviate positive as POS and then plus false positives, again abbreviating positive using POS. So that's called precision, and as you can tell high precision would be good. That means that all the patients that we went to and we said, "You know, we're very sorry. We think you have cancer," high precision means that of that group of patients most of them we had actually made accurate predictions on them and they do have cancer.

The second number we're going to compute is called recall, and what recall say is, if all the patients in, let's say, in the test set or the cross-validation set, but if all the patients in the data set that actually have cancer, what fraction of them that we correctly detect as having cancer. So if all the patients have cancer, how many of them did we actually go to them and you know, correctly told them that we think they need treatment. So, writing this down, recall is defined as the number of positives, excuse me, the number of true positives, meaning the number of people that have cancer and that we correctly predicted have cancer and we take that and divide that by, divide that by the number of actual positives, so this is the right number of actual positives of all the people that do have cancer. What fraction do we directly flag and you know, set the treatment.

So, to rewrite this in a different form, the denominator would be the number of actual positives as you know, is the sum of the entries in this first column over here. And so writing things out differently, this is therefore, the number of true positives, divided by the number of true positives plus the number of false negatives. And so once again, having a high recall would be a good thing.

So by computing precision and recall this will usually give us a better sense of how well our classifier is doing. And in particular if we have a learning algorithm that predicts  $y$  equals zero all the time, if it predicts no one has cancer, then this classifier will have a recall equal to zero, because there won't be any true positives and so that's a quick way for us to recognize that, you know, a classifier that predicts  $y$  equals 0 all the time, just isn't a very good classifier.

And more generally, even for settings where we have very skewed classes, it's not possible for an algorithm to sort of "cheat" and somehow get a very high precision and a very high recall by doing some simple thing like predicting  $y$  equals 0 all the time or predicting  $y$  equals 1 all the time. And so we're much more sure that a classifier of a high precision or high recall actually is a good classifier, and this gives us a more useful evaluation metric that is a more direct way to actually understand whether, you know, our algorithm may be doing well.

So one final note in the definition of precision and recall, that we would define precision and recall, usually we use the convention that  $y$  is equal to 1, in the presence of the more rare class. So if we are trying to detect some rare conditions such as cancer, hopefully that's a rare condition, precision and recall are defined setting  $y$  equals 1, rather than  $y$  equals 0, to be sort of presence of that rare class that we're trying to detect.

And by using precision and recall, we find, what happens is that even if we have very skewed classes, it's not possible for an algorithm to you know, "cheat" and predict  $y$  equals 1 all the time, or predict  $y$  equals 0 all the time, and get high precision and recall. And in particular, if a classifier is getting high precision and high recall, then we are actually confident that the algorithm has to be doing well, even if we have very skewed classes.

So for the problem of skewed classes precision recall gives us more direct insight into how the learning algorithm is doing and this is often a much better way to evaluate our learning algorithms, than looking at classification error or classification accuracy, when the classes are very skewed.

## Trading Off Precision and Recall

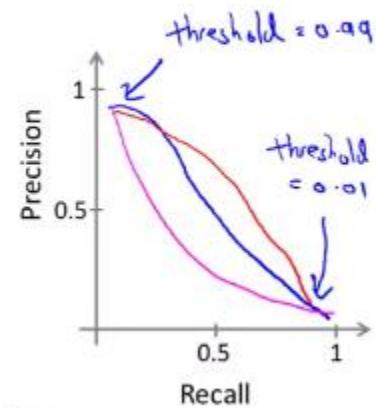
In the last video, we talked about precision and recall as an evaluation metric for classification problems with skewed classes.

For many applications, we'll want to somehow control the trade-off between precision and recall. Let me tell you how to do that and also show you some even more effective ways to use precision and recall as an evaluation metric for learning algorithms.

- **Trading off precision and recall**
- Logistic regression:  $0 \leq h_\theta(x) \leq 1$
- Predict 1 if  $h_\theta(x) \geq 0.8$  ~~0.7~~ ~~0.9~~ ~~0.3~~ ↗
- Predict 0 if  $h_\theta(x) < 0.8$  ~~0.7~~ ~~0.9~~ ~~0.3~~
- Suppose we want to predict  $y = 1$  (cancer) only if very confident.  
    → Higher precision, lower recall.
- Suppose we want to avoid missing too many cases of cancer (avoid false negatives).  
    → Higher recall, lower precision.

$$\rightarrow \text{precision} = \frac{\text{no. of predicted positive}}{\text{true positives}}$$

$$\rightarrow \text{recall} = \frac{\text{true positives}}{\text{no. of actual positive}}$$



More generally: Predict 1 if  $h_\theta(x) \geq \text{threshold}$ . ↗

As a reminder, here are the definitions of precision and recall from the previous video. Let's continue our cancer classification example, where  $y$  equals 1 if the patient has cancer and  $y$  equals 0 otherwise. And let's say we're trained in logistic regression classifier which outputs probability between 0 and 1. So, as usual, we're going to predict 1,  $y$  equals 1, if  $h(x)$  is greater or

equal to 0.5. And predict 0 if the hypothesis outputs a value less than 0.5. And this classifier may give us some value for precision and some value for recall.

But now, suppose we want to predict that the patient has cancer only if we're very confident that they really do. Because if you go to a patient and you tell them that they have cancer, it's going to give them a huge shock. What we give is a seriously bad news, and they may end up going through a pretty painful treatment process and so on. And so maybe we want to tell someone that we think they have cancer only if they are very confident.

One way to do this would be to modify the algorithm, so that instead of setting this threshold at 0.5, we might instead say that we will predict that  $y$  is equal to 1 only if  $h(x)$  is greater or equal to 0.7. So this is like saying, we'll tell someone they have cancer only if we think there's a greater than or equal to, 70% chance that they have cancer. And, if you do this, then you're predicting someone has cancer only when you're more confident and so you end up with a classifier that has higher precision. Because all the patients that you're going to and saying, we think you have cancer, although those patients are now ones that you're pretty confident actually have cancer.

And so a higher fraction of the patients that you predict have cancer will actually turn out to have cancer because making those predictions only if we're pretty confident. But in contrast this classifier will have lower recall because now we're going to make predictions, we're going to predict  $y = 1$  on a smaller number of patients. Now, can even take this further. Instead of setting the threshold at 0.7, we can set this at 0.9. Now we'll predict  $y=1$  only if we are more than 90% certain that the patient has cancer. And so, a large fraction of those patients will turn out to have cancer. And so this would be a higher precision classifier will have lower recall because we want to correctly detect that those patients have cancer.

Now consider a different example. Suppose we want to avoid missing too many actual cases of cancer, so we want to avoid false negatives. In particular, if a patient actually has cancer, but we fail to tell them that they have cancer then that can be really bad. Because if we tell a patient that they don't have cancer, then they're not going to go for treatment. And if it turns out that they have cancer, but we fail to tell them they have cancer, well, they may not get treated at all. And so that would be a really bad outcome because they die because we told them that they don't have cancer. They fail to get treated, but it turns out they actually have cancer. So, suppose that, when in doubt, we want to predict that  $y=1$ . So, when in doubt, we want to predict that they have cancer so that at least they look further into it, and these can get treated in case they do turn out to have cancer.

In this case, rather than setting higher probability threshold, we might instead take this value and instead set it to a lower value. So maybe 0.3 like so, right? And by doing so, we're saying that, you know what, if we think there's more than a 30% chance that they have cancer we better be more conservative and tell them that they may have cancer so that they can seek treatment if necessary. And in this case what we would have is going to be a higher recall classifier, because we're going to be correctly flagging a higher fraction of all of the patients that actually do have cancer. But we're going to end up with lower precision because a higher fraction of the patients that we said have cancer, a high fraction of them will turn out not to have cancer after all.

And by the way, just as a sider, when I talk about this to other students, I've been told before, it's pretty amazing, some of my students say, is how I can tell the story both ways. Why we might want to have higher precision or higher recall and the story actually seems to work both ways.

But I hope the details of the algorithm is true and the more general principle is depending on where you want, whether you want higher precision- lower recall, or higher recall- lower precision. You can end up predicting  $y=1$  when  $h(x)$  is greater than some threshold. And so in general, for most classifiers there is going to be a trade off between precision and recall, and as you vary the value of this threshold that we join here, you can actually plot out some curve that trades off precision and recall where a value up here, this would correspond to a very high value of the threshold, maybe threshold equals 0.99. So that's saying, predict  $y=1$  only if we're more than 99% confident, at least 99% probability this one. So that would be a high precision, relatively low recall. Where as the point down here, will correspond to a value of the threshold that's much lower, maybe equal 0.01, meaning, when in doubt at all, predict  $y=1$ , and if you do that, you end up with a much lower precision, higher recall classifier. And as you vary the threshold, if you want you can actually trace of a curve for your classifier to see the range of different values you can get for precision recall.

And by the way, the precision-recall curve can look like many different shapes. Sometimes it will look like this, sometimes it will look like that. Now there are many different possible shapes for the precision-recall curve, depending on the details of the classifier.

So, this raises another interesting question which is, is there a way to choose this threshold automatically?

### **F<sub>1</sub> Score (F score)**

How to compare precision/recall numbers?

	Precision(P)	Recall (R)	Average	F <sub>1</sub> Score
Algorithm 1	0.5	0.4	0.45	0.444 ←
Algorithm 2	0.7	0.1	0.4	0.175 ←
Algorithm 3	0.02	1.0	0.51	0.0392 ←

Average:  $\frac{P+R}{2}$

$F_1 \text{ Score: } 2 \frac{PR}{P+R}$

P = 0 or R = 0  $\Rightarrow F\text{-score} = 0$ .

P = 1 and R = 1  $\Rightarrow F\text{-score} = 1$ .

Or more generally, if we have a few different algorithms or a few different ideas for algorithms, how do we compare different precision recall numbers?

Concretely, suppose we have three different learning algorithms. So actually, maybe these are three different learning algorithms, maybe these are the same algorithm but just with different values for the threshold. How do we decide which of these algorithms is best? One of the things

we talked about earlier is the importance of a single real number evaluation metric. And that is the idea of having a number that just tells you how well is your classifier doing. But by switching to the precision recall metric we've actually lost that. We now have two real numbers. And so we often, we end up face the situations like if we trying to compare Algorithm 1 and Algorithm 2, we end up asking ourselves, is the precision of 0.5 and a recall of 0.4, was that better or worse than a precision of 0.7 and recall of 0.1? And, if every time you try out a new algorithm you end up having to sit around and think, well, maybe  $0.5/0.4$  is better than  $0.7/0.1$ , or maybe not, I don't know. If you end up having to sit around and think and make these decisions, that really slows down your decision making process for what things, what changes are useful to incorporate into your algorithm.

Whereas in contrast, if we have a single real number evaluation metric like a number that just tells us is algorithm 1 or is algorithm 2 better, then that helps us to much more quickly decide which algorithm to go with. It helps us as well to much more quickly evaluate different changes that we may be contemplating for an algorithm.

So how can we get a single real number evaluation metric? One natural thing that you might try is to look at the average precision and recall. So, using P and R to denote precision and recall, what you could do is just compute the average and look at what classifier has the highest average value. But this turns out not to be such a good solution, because similar to the example we had earlier it turns out that if we have a classifier that predicts  $y=1$  all the time, then if you do that you can get a very high recall, but you end up with a very low value of precision.

Conversely, if you have a classifier that predicts  $y$  equals zero, almost all the time, that is that it predicts  $y=1$  very sparingly, this corresponds to setting a very high threshold using the notation of the previous  $y$ . Then you can actually end up with a very high precision with a very low recall. So, the two extremes of either a very high threshold or a very low threshold, neither of that will give a particularly good classifier. And the way we recognize that is by seeing that we end up with a very low precision or a very low recall. And if you just take the average of  $(P+R)/2$  from this example, the average is actually highest for Algorithm 3, even though you can get that sort of performance by predicting  $y=1$  all the time and that's just not a very good classifier, right? You predict  $y=1$  all the time, just normal useful classifier, but all it does is prints out  $y=1$ . And so Algorithm 1 or Algorithm 2 would be more useful than Algorithm 3. But in this example, Algorithm 3 has a higher average value of precision recall than Algorithms 1 and 2.

So we usually think of this average of precision and recall as not a particularly good way to evaluate our learning algorithm.

In contrast, there's a different way for combining precision and recall. This is called the F Score and it uses that formula. And so in this example, here are the F Scores. And so we would tell from these F Scores, it looks like Algorithm 1 has the highest F Score, Algorithm 2 has the second highest, and Algorithm 3 has the lowest. And so, if we go by the F Score we would pick probably Algorithm 1 over the others. The F Score, which is also called the F1 Score, is usually written F1 Score that I have here, but often people will just say F Score, either term is used. Is a little bit like taking the average of precision and recall, but it gives the lower value of precision and recall, whichever it is, it gives it a higher weight. And so, you see in the numerator here that

the F Score takes a product of precision and recall. And so if either precision is 0 or recall is equal to 0, the F Score will be equal to 0.

So in that sense, it kind of combines precision and recall, but for the F Score to be large, both precision and recall have to be pretty large. I should say that there are many different possible formulas for combining precision and recall. This F Score formula is really maybe a, just one out of a much larger number of possibilities, but historically or traditionally this is what people in Machine Learning seem to use. And the term F Score, it doesn't really mean anything, so don't worry about why it's called F Score or F1 Score.

But this usually gives you the effect that you want because if either a precision is zero or recall is zero, this gives you a very low F Score, and so to have a high F Score, you kind of need a precision or recall to be one. And concretely, if  $P=0$  or  $R=0$ , then this gives you that the F Score = 0. Whereas a perfect F Score, so if precision equals one and recall equals 1, that will give you an F Score, that's equal to 1 times 1 over 2 times 2, so the F Score will be equal to 1, if you have perfect precision and perfect recall. And intermediate values between 0 and 1, this usually gives a reasonable rank ordering of different classifiers.

So in this video, we talked about the notion of trading off between precision and recall, and how we can vary the threshold that we use to decide whether to predict  $y=1$  or  $y=0$ . So it's the threshold that says, do we need to be at least 70% confident or 90% confident, or whatever before we predict  $y=1$ . And by varying the threshold, you can control a trade off between precision and recall. We also talked about the F Score, which takes precision and recall, and again, gives you a single real number evaluation metric. And of course, if your goal is to automatically set that threshold to decide what's really  $y=1$  and  $y=0$ , one pretty reasonable way to do that would also be to try a range of different values of thresholds.

So you try a range of values of thresholds and evaluate these different thresholds on, say, your cross-validation set and then to pick whatever value of threshold gives you the highest F Score on your cross-validation set. And that is a pretty reasonable way to automatically choose the threshold for your classifier as well.

## Data for Machine Learning

In the previous video, we talked about evaluation metrics.

In this video, I'd like to switch tracks a bit and touch on another important aspect of machine learning system design, which will often come up, which is the issue of how much data to train on.

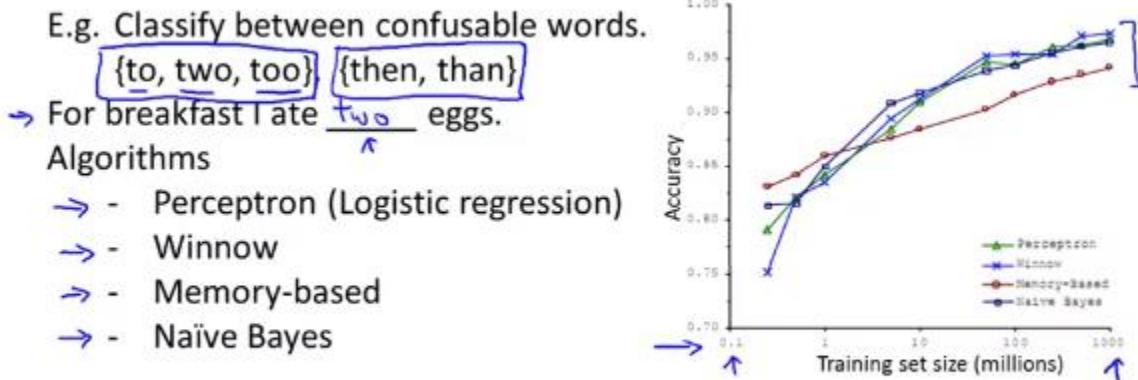
Now, in some earlier videos, I had cautioned against blindly going out and just spending lots of time collecting lots of data, because it's only sometimes that that would actually help. But it turns out that under certain conditions, and I will say in this video what those conditions are, getting a lot of data and training on a certain type of learning algorithm, can be a very effective way to get a learning algorithm to do very good performance.

And this arises often enough that if those conditions hold true for your problem and if you're able to get a lot of data, this could be a very good way to get a very high performance learning algorithm.

So in this video, let's talk more about that.

Let me start with a story.

### Designing a high accuracy learning system



**"It's not who has the best algorithm that wins.**

**It's who has the most data."**

Many, many years ago, two researchers that I know, Michelle Banko and Eric Brodale ran the following fascinating study. They were interested in studying the effect of using different learning algorithms versus trying them out on different training set sizes. They were considering the problem of classifying between confusable words, so for example, in the sentence: for breakfast I ate, should it be to, two or too? Well, for this example, for breakfast I ate two, 2 eggs. So, this is one example of a set of confusable words and that's a different set. So they took machine learning problems like these, sort of supervised learning problems to try to categorize what is the appropriate word to go into a certain position in an English sentence.

They took a few different learning algorithms which were, you know, sort of considered state of the art back in the day, when they ran the study in 2001, so they took a variance, roughly a variance on logistic regression called the Perceptron. They also took some of their algorithms that were fairly out back then but somewhat less used now so when the algorithm also very similar to which is a regression but different in some ways, much used somewhat less, used not too much right now took what's called a memory based learning algorithm again used somewhat less now. But I'll talk a little bit about that later. And they used a naive based algorithm, which is something they'll actually talk about in this course. The exact algorithms of these details aren't important. Think of this as, you know, just picking four different classification algorithms and really the exact algorithms aren't important. But what they did was they varied the training set size and tried out these learning algorithms on the range of training set sizes and that's the result they got.

And the trends are very clear right. First most of these algorithms give remarkably similar performance. And second, as the training set size increases, on the horizontal axis is the training

set size in millions, as you go from you know a hundred thousand up to a thousand million that is a billion training examples. The performance of the algorithms all pretty much monotonically increase and the fact that if you pick any algorithm may be pick a "inferior algorithm" but if you give that "inferior algorithm" more data, then from these examples, it looks like it will most likely beat even a "superior algorithm".

So since this original study which is very influential, there's been a range of many different studies showing similar results that show that many different learning algorithms you know tend to, can sometimes, depending on details, can give pretty similar ranges of performance, but what can really drive performance is you can give the algorithm a ton of training data. And this is, results like these has led to a saying in machine learning that often in machine learning it's not who has the best algorithm that wins, it's who has the most data. So when is this true and when is this not true?

Because we have a learning algorithm for which this is true then getting a lot of data is often maybe the best way to ensure that we have an algorithm with very high performance rather than you know, debating worrying about exactly which of these algorithms to use.

### **Large data rationale**

→ Assume feature  $x \in \mathbb{R}^{n+1}$  has sufficient information to predict  $y$  accurately.

Example: For breakfast I ate two eggs.

Counterexample: Predict housing price from only size (feet<sup>2</sup>) and no other features.

Useful test: Given the input  $x$ , can a human expert confidently predict  $y$ ?

Let's try to lay out a set of assumptions under which having a massive training set we think will be able to help. Let's assume that in our machine learning problem, the features  $x$  have sufficient information with which we can use to predict  $y$  accurately. For example, if we take the confusable words problem that we had on the previous slide. Let's say that it features  $x$  capture what are the surrounding words, around the blank that we're trying to fill in. So the features capture, that we want to have, sentence "For breakfast I ate \_\_\_\_ eggs". Then yeah that is pretty much information to tell me that the word I want in the middle is TWO and that is not word TO and its not the word TOO. So the features capture, you know, one of these surrounding words then that gives me enough information to pretty unambiguously decide what is the label  $y$  or in other words what is the word that I should be using to fill in that blank out of this set of three confusable words. So that's an example of what the features  $x$  have sufficient information for specific  $y$ .

For a counter example. Consider a problem of predicting the price of a house from only the size of the house and from no other features. So if you imagine I tell you that a house is, you know, 500 square feet but I don't give you any other features. I don't tell you that the house is in an expensive part of the city. Or if I don't tell you that the house, the number of rooms in the house, or how nicely furnished the house is, or whether the house is new or old. If I don't tell you anything other than that this is a 500 square foot house, well there's so many other factors that would affect the price of a house other than just the size of a house that if all you know is the size it's actually very difficult to predict the price accurately.

So that would be a counter example to this assumption that the features have sufficient information to predict the price to the desired level of accuracy. The way I think about testing this assumption, one way I often think about it is, how often I ask myself. Given the input features  $x$ , given the features, given the same information available as well as learning algorithm. If we were to go to human expert in this domain. Can a human experts actually or can human expert confidently predict the value of  $y$ .

For this first example if we go to, you know an expert human English speaker. You go to someone that speaks English well, right, then a human expert in English just read most people like you and me, will probably we would probably be able to predict what word should go in here. Till a good English speaker can predict this well, and so this gives me confidence that  $x$  allows us to predict  $y$  accurately. But in contrast if we go to an expert in human prices, like maybe an expert realtor, right, someone who sells houses for a living. If I just tell them the size of a house and I tell them what the price is well even an expert in pricing or selling houses wouldn't be able to tell me and so this is fine that for the housing price example knowing only the size doesn't give me enough information to predict the price of the house.

So, let's say, this assumption holds. Let's see then, when having a lot of data could help. Suppose the features have enough information to predict the value of  $y$ . And let's suppose we use a learning algorithm with a large number of parameters, so maybe logistic regression or linear regression with a large number of features. Or one thing that I sometimes do, one thing that I often do actually is using neural network with many hidden units that would be another learning algorithm with a lot of parameters.

==

So these are all powerful learning algorithms with a lot of parameters that can fit very complex functions. So, I'm going to call these, I'm going to think of these as low-bias algorithms because you know we can fit very complex functions and because we have a very powerful learning algorithm that can fit very complex functions. Chances are, if we run these algorithms on the data sets, it will be able to fit the training set well, and so hopefully the training error will be low.

Now let's say, we use a massive, massive training set. In that case, if we have a huge training set, then hopefully even though we have a lot of parameters but if the training set is sort of even much larger than the number of parameters then hopefully these algorithms will be unlikely to overfit. Right because we have such a massive training set. And by unlikely to overfit what that means is that the training error will hopefully be close to the test error. Finally putting these two together that the train set error is small and the test set error is close to the training error what these two together imply is that hopefully the test set error will also be small.

Another way to think about this is that in order to have a high performance learning algorithm we want it not to have high bias and not to have high variance.

So the bias problem we're going to address by making sure we have a learning algorithm with **many parameters** and so that gives us a **low bias** algorithm and by using a **very large training set**, this ensures that we don't have a **variance problem** either. So hopefully our algorithm will have low variance and so is by putting these two together, that we end up with a low bias and a low variance learning algorithm, and this allows us to do well on the test set.

And fundamentally it's the key ingredients of assuming that the **features have enough information** and we have a rich class of functions that's what guarantees **low bias**, and then it having a **massive training set** that that's what guarantees **more variance**.

So this gives us a set of conditions rather hopefully some understanding of what's the sort of problem where if you have a lot of data and you train a learning algorithm with a lot of parameters, that might be a good way to give a high performance learning algorithm and really, I think the key test that I often ask myself are first, can a human expert look at the features  $x$  and confidently predict the value of  $y$ . Because that's sort of a certification that  $y$  can be predicted accurately from the features  $x$  and second, can we actually get a large training set, and train the learning algorithm with a lot of parameters in the training set and if you can do both then that's what often will give you a very high performance learning algorithm.

## Support Vector Machines

### Large Margin Classification

#### Optimization Objective

By now, you've seen a range of different learning algorithms. With supervised learning, the performance of many supervised learning algorithms will be pretty similar, and what matters less often will be whether you use learning algorithm a or learning algorithm b, but what matters more will often be things like the amount of data you create these algorithms on, as well as your skill in applying these algorithms.

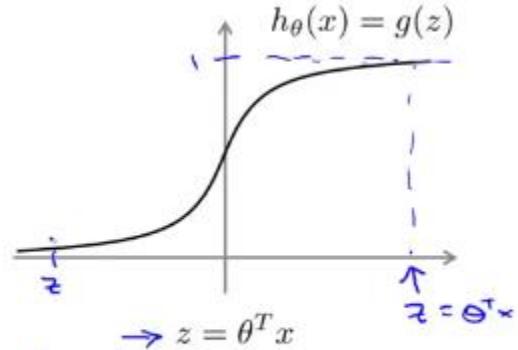
Things like your choice of the features you design to give to the learning algorithms, and how you choose the regularization parameter, and things like that.

But, there's one more algorithm that is very powerful and is very widely used both within industry and academia, and that's called the support vector machine. And compared to both logistic regression and neural networks, the **Support Vector Machine, or SVM sometimes gives a cleaner, and sometimes more powerful way of learning complex non-linear functions**. And so let's take the next videos to talk about that. Later in this course, I will do a quick survey of a range of different supervisory learning algorithms just as very briefly describe them. But the support vector machine, given its popularity and how powerful it is, this will be the last of the supervisory algorithms that I'll spend a significant amount of time on in this course. As with our

development of other learning algorithms, we're gonna start by talking about the optimization objective. So, let's get started on this algorithm.

## Alternative view of logistic regression

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



If  $y = 1$ , we want  $h_{\theta}(x) \approx 1$ ,  $\theta^T x \gg 0$   
 If  $y = 0$ , we want  $h_{\theta}(x) \approx 0$ ,  $\theta^T x \ll 0$

In order to describe the support vector machine, I'm actually going to start with logistic regression, and show how we can modify it a bit, and get what is essentially the support vector machine. So in logistic regression, we have our familiar form of the hypothesis there and the sigmoid activation function shown on the right. And in order to explain some of the math, I'm going to use z to denote  $\theta^T x$  here.

Now let's think about what we would like logistic regression to do. If we have an example with y equals 1 and by this I mean an example in either the training set or the test set or the cross-validation set, but when y is equal to 1 then we're sort of hoping that  $h(x)$  will be close to 1. Right, we're hoping to correctly classify that example. And what having  $h(x)$  close to 1, what that means is that  $\theta^T x$  must be much larger than 0. So there's greater than, greater than sign that means much, much greater than 0. And that's because it is z, that is this  $\theta^T x$  is, when z is much bigger than 0, is far to the right of this figure that the output of logistic regression becomes close to one.

Conversely, if we have an example where y is equal to zero, then what we're hoping for is that the hypothesis will output a value close to zero. And that corresponds to  $\theta^T x$  or z being much less than zero because that corresponds to hypothesis of putting a value close to zero.

## Alternative view of logistic regression

$$\begin{aligned} \text{Cost of example: } & -(y \log h_{\theta}(x) + (1 - y) \log(1 - h_{\theta}(x))) \leftarrow \\ & = -y \log \frac{1}{1 + e^{-\theta^T x}} - (1 - y) \log \left(1 - \frac{1}{1 + e^{-\theta^T x}}\right) \leftarrow \end{aligned}$$

If you look at the cost function of logistic regression, what you'll find is that each example (x,y) contributes a term like this to the overall cost function, right? So for the overall cost function, we usually, we will also have a sum over all the training examples and the  $1/m$  term, that this

expression here, that's the term that a single training example contributes to the overall objective function for logistic regression.

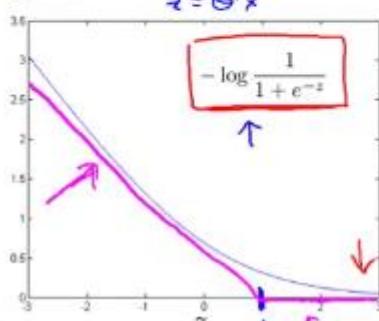
Now if I take the definition for the form of my hypothesis and plug it in over here, then what I get is that each training example contributes this term, ignoring the one/m but it contributes that term to my overall cost function for logistic regression.

### Alternative view of logistic regression (x, y)

$$\text{Cost of example: } -(y \log h_{\theta}(x) + (1 - y) \log(1 - h_{\theta}(x))) \leftarrow$$

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

If  $y = 1$  (want  $\theta^T x \gg 0$ ):



Now let's consider the two cases of when  $y$  is equal to 1 and when  $y$  is equal to 0. In the first case, let's suppose that  $y$  is equal to 1. In that case, only this first term in the objective matters, because this one minus  $y$  term would be equal to 0 if  $y$  is equal to 1. So when  $y$  is equal to 1, when in our example  $(x, y)$ , when  $y$  is equal to 1 what we get is this term, minus log one over one plus  $e$  to the negative  $z$ , where as similar to the last slide I'm using  $z$  to denote  $\theta^T x$  and of course we actually have this minus  $y$ , but we will just add if  $y$  is equal to one so that's equal to one I just simplify it in a way in the expression that I have written down here.

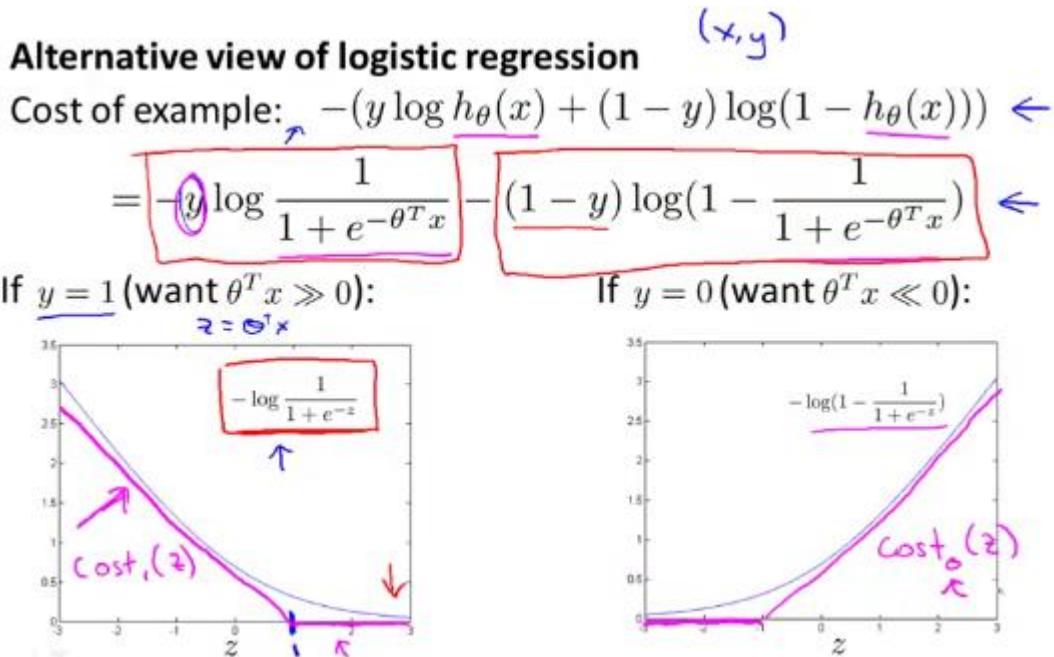
And if we plot this function as a function of  $z$ , what you find is that you get this curve shown on the lower left of the slide. And thus, we also see that when  $z$  is equal to large, that is, when  $\theta^T x$  is large, that corresponds to a value of  $z$  that gives us a fairly small value, a very, very small contribution to the cost function.

And this kinda explains why, when logistic regression sees a positive example, with  $y=1$ , it tries to set  $\theta^T x$  to be very large because that corresponds to this term, and the cost function being small.

Now, to fill the support vector machine, here's what we're going to do. We're gonna take this cost function, this minus log 1 over 1 plus  $e$  to negative  $z$ , and modify it a little bit. Let me take this point, one over here, and let me draw the cost function we're going to use.

The new cost functions can be flat from here on out, and then I am gonna draw something that grows as a straight line, similar to logistic regression but this is going to be a straight line at this

portion. So the curve that I just drew in magenta, and the curve I just drew purple and magenta, so if it's pretty close approximation to the cost function used by logistic regression. Except that it is now made up of two line segments, there's this flat portion on the right, and then there's this straight line portion on the left. And don't worry too much about the slope of the straight line portion. It doesn't matter that much. But that's the new cost function we're going to use for when  $y$  is equal to one, and you can imagine it should do something pretty similar to logistic regression. But turns out, that this will give the support vector machine computational advantages and give us, later on, an easier optimization problem that would be easier for software to solve.



We just talked about the case of  $y$  equals one. The other case is if  $y$  is equal to zero. In that case, if you look at the cost, then only the second term will apply because the first term goes away, right? If  $y$  is equal to zero, then you have a zero here, so you're left only with the second term of the expression above. And so the cost of an example, or the contribution of the cost function, is going to be given by this term over here. And if you plot that as a function of  $z$ , so I have pure  $z$  on the horizontal axis, you end up with this curve. And for the support vector machine, once again, we're going to replace this blue line with something similar and at the same time we replace it with a new cost, this flat out here, this 0 out here. And that then grows as a straight line, like so. So let me give these two functions names. This function on the left I'm going to call  $\text{Cost}_1(z)$ , and this function of the right I'm gonna call  $\text{Cost}_0(z)$ . And the subscript just refers to the cost corresponding to when  $y$  is equal to 1, versus when  $y$  is equal to zero. Armed with these definitions, we're now ready to build a support vector machine.

### Support vector machine

Logistic regression:

$$\min_{\theta} \frac{1}{m} \left[ \sum_{i=1}^m y^{(i)} \underbrace{\left( -\log h_\theta(x^{(i)}) \right)}_{\text{Cost}_1(\theta^T x^{(i)})} + (1 - y^{(i)}) \underbrace{\left( -\log(1 - h_\theta(x^{(i)})) \right)}_{\text{Cost}_0(\theta^T x^{(i)})} \right] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

Here's the cost function,  $J(\theta)$  that we have for logistic regression. In case this equation looks a bit unfamiliar, it's because previously we had a minus sign outside, but here what I did was I instead moved the minus signs inside these expressions, so it just makes it look a little different.

For the support vector machine what we're going to do is essentially take this and replace this with  $\text{cost}_1(z)$ , that is  $\text{cost}_1(\theta^T x^{(i)})$ . And we're going to take this and replace it with  $\text{cost}_0(z)$ , that is  $\text{cost}_0(\theta^T x^{(i)})$ . Where the cost one function is what we had on the previous slide that looks like this. And the cost zero function, again what we had on the previous slide, and it looks like this.

**Support vector machine:**

$$\min_{\theta} \frac{1}{m} \sum_{i=1}^m y^{(i)} \text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)}) \text{cost}_0(\theta^T x^{(i)}) + \frac{\lambda}{2m} \sum_{j=0}^n \theta_j^2$$

So what we have for the support vector machine is a minimization problem of one over m, the sum over my training examples of  $y^{(i)}$  times  $\text{cost}_1(\theta^T x^{(i)})$ , plus one minus  $y^{(i)}$  times  $\text{cost}_0(\theta^T x^{(i)})$ , and then plus my usual regularization parameter, like so.

**Support vector machine**

**Logistic regression:**

$$\min_{\theta} \frac{1}{m} \left[ \sum_{i=1}^m y^{(i)} \underbrace{\left( -\log h_{\theta}(x^{(i)}) \right)}_{\text{cost}_1(\theta^T x^{(i)})} + (1-y^{(i)}) \underbrace{\left( -\log(1-h_{\theta}(x^{(i)})) \right)}_{\text{cost}_0(\theta^T x^{(i)})} \right] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

**Support vector machine:**

$$\min_{\theta} \cancel{\frac{1}{m}} C \sum_{i=1}^m y^{(i)} \text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)}) \text{cost}_0(\theta^T x^{(i)}) + \frac{1}{2} \cancel{\lambda} \sum_{j=0}^n \theta_j^2$$

$$\min_u \frac{(u-5)^2 + 1}{10} \rightarrow u=5 \quad | \quad \cancel{A} + \cancel{\lambda} \cancel{B} \leftarrow \quad C = \frac{1}{\cancel{\lambda}}$$

$$\min_u 10(u-5)^2 + 10 \rightarrow u=5 \quad | \quad \cancel{C} \cancel{A} + \cancel{B} \leftarrow$$

$$\Rightarrow \min_{\theta} C \sum_{i=1}^m \left[ y^{(i)} \text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)}) \text{cost}_0(\theta^T x^{(i)}) \right] + \frac{1}{2} \sum_{j=0}^n \theta_j^2$$

Now, by convention, for the support vector machine, we actually write things slightly differently. We re-parameterize this just very slightly differently. First, we're going to get rid of the 1 over m terms, and this just happens to be a slightly different convention that people use for support vector machines compared to logistic regression. Here's what I mean.

Your one way to do this, I'm just gonna get rid of these one over m terms and this should give you me the same optimal value of  $\theta$ , right? Because one over m is just a constant, so whether I solve this minimization problem with one over m in front or not I should end up with the same optimal value for  $\theta$ .

Here's what I mean, to give you a concrete example, suppose I had a minimization problem, minimize over a number  $u$  of  $(u-5)^2 + 1$ . Right, well, the minimum of this happens to be  $u = 5$ . Now if I were to take this objective function and multiply it by 10, so here my minimization problem is  $\min_u 10(u-5)^2 + 10$ . Well the value of  $u$  that minimizes this is still  $u = 5$ , right? So multiplying something that you're minimizing over, by some constant, 10 in this case, it does not change the value of  $u$  that gives us, that minimizes this function.

So the same way, what I've done by crossing out this  $m$  is all I'm doing is multiplying my objective function by some constant  $m$  and it doesn't change the value of  $\theta$  that achieves the minimum.

The second bit of notational change, which is just, again, the more standard convention when using SVMs instead of logistic regression, is the following. So for logistic regression, we had two terms to the objective function. The first is this term, which is the cost that comes from the training set and the second is this term, which is the regularization term.

And what we had was we had  $A$ , we control the trade-off between these by saying, we want to minimize  $A$  plus, and then my regularization parameter  $\lambda$ , and then times some other term  $B$ , where I guess I'm using your  $A$  to denote this first term, and I'm using  $B$  to denote that second term, maybe without the  $\lambda$ .

And instead of parameterizing this as  $A$  plus  $\lambda B$ , we could, and so what we did was by setting different values for this regularization parameter  $\lambda$ , we could trade off the relative weight between how much we wanted the training set well, that is, minimizing  $A$ , versus how much we care about keeping the values of the parameter small, so that will be parameter  $B$ .

For the support vector machine, just by convention, we're going to use a different parameter. So instead of using  $\lambda$  here to control the relative weighting between the first and second terms, we're instead going to use a different parameter which by convention is called  $C$  and is set to minimize  $C$  times  $A + B$ .

So for logistic regression, if we set a very large value of  $\lambda$  that means you will give  $B$  a very high weight. Here is that if we set  $C$  to be a very small value, then that corresponds to giving  $B$  a much larger weight than  $C$ , than  $A$ . So this is just a different way of controlling the trade off, it's just a different way of parameterizing how much we care about optimizing the first term, versus how much we care about optimizing the second term. And if you want you can think of this as the parameter  $C$  playing a role similar to  $1/\lambda$ . And it's not that it's two equations or these two expressions will be equal, this equals  $1/\lambda$ , that's not the case. It's rather that if  $C$  is equal to  $1/\lambda$ , then these two optimization objectives should give you the same value the same optimal value for  $\theta$  so we just fill in that in I'm gonna cross out  $\lambda$  here and write in the constant  $C$  there. So that gives us our overall optimization objective function for the support vector machine. And if you minimize that function, then what you have is the parameters learned by the SVM.

## SVM hypothesis

$$\Rightarrow \min_{\theta} C \sum_{i=1}^m \left[ y^{(i)} \text{cost}_1(\theta^T x^{(i)}) + (1 - y^{(i)}) \text{cost}_0(\theta^T x^{(i)}) \right] + \frac{1}{2} \sum_{j=1}^n \theta_j^2$$

Hypothesis:

$$h_{\theta}(x) = \begin{cases} 1 & \text{if } \theta^T x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Finally, unlike logistic regression, the support vector machine doesn't output the probability is that what we have is we have this cost function, which we minimize to get the parameter's  $\theta$ , and what a support vector machine does is it just makes a prediction of  $y$  being equal to one or zero, directly. So the hypothesis will predict one if  $\theta^T x$  is greater than or equal to zero, and it will predict zero otherwise and so having learned the parameters  $\theta$ , this is the form of the hypothesis for the support vector machine.

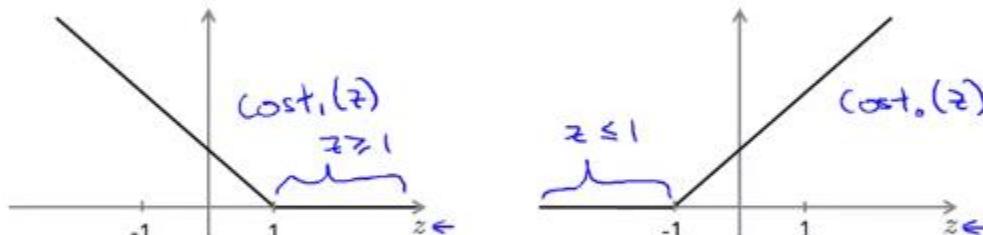
So that was a mathematical definition of what a support vector machine does. In the next few videos, let's try to get back to intuition about what this optimization objective leads to and whether the source of the hypotheses SVM will learn and we'll also talk about how to modify this just a little bit to the complex nonlinear functions.

### Large Margin Intuition

Sometimes people talk about support vector machines as large margin classifiers. In this video I'd like to tell you what that means, and this will also give us a useful picture of what an SVM hypothesis may look like.

### Support Vector Machine

$$\rightarrow \min_{\theta} C \sum_{i=1}^m \left[ y^{(i)} \underline{\text{cost}_1(\theta^T x^{(i)})} + (1 - y^{(i)}) \underline{\text{cost}_0(\theta^T x^{(i)})} \right] + \frac{1}{2} \sum_{j=1}^n \theta_j^2$$



→ If  $y = 1$ , we want  $\underline{\theta^T x} \geq 1$  (not just  $\geq 0$ )

→ If  $y = 0$ , we want  $\underline{\theta^T x} \leq -1$  (not just  $< 0$ )

$$\begin{aligned} \theta^T x &\geq 1 \\ \theta^T x &\leq -1 \end{aligned}$$

$$C = 100,000$$

Here's my cost function for the support vector machine where here on the left I've plotted my  $\text{Cost}_1(z)$  function that I used for positive examples, and on the right I've plotted my  $\text{Cost}_0(z)$  function, where I have  $z$  here on the horizontal axis.

Now, let's think about what it takes to make these cost functions small. If you have a positive example, so if  $y$  is equal to 1, then  $\text{Cost}_1(z)$  is zero only when  $z$  is greater than or equal to 1. So in other words, if you have a positive example, we really want  $\theta^T x$  to be greater than or equal to 1, and conversely if  $y$  is equal to zero, look this  $\text{Cost}_0(z)$  function, then it's only in this region where  $z$  is less than equal to 1 we have the cost is zero as  $z$  is equals to zero. And this is an interesting property of the support vector machine right, which is that, if you have a positive example so if  $y$  is equal to one, then all we really need is that  $\theta^T x$  is greater than equal to zero, and that would mean that we classify correctly because if  $\theta^T x$  is greater than zero our hypothesis will predict zero.

And similarly, if you have a negative example, then really all you want is that  $\theta^T x$  is less than zero and that will make sure we got the example right. But the support vector machine wants a bit more than that. It says, you know, don't just barely get the example right. So then don't just have it just a little bit bigger than zero. What I really want is for this to be quite a lot bigger than zero say maybe bit greater or equal to one and I want this to be much less than zero. Maybe I want it less than or equal to -1. And so this builds in an extra safety factor or safety margin factor into the support vector machine. Logistic regression does something similar too of course, but let's see what happens or let's see what the consequences of this are, in the context of the support vector machine.

Concretely, what I'd like to do next is consider a case where we set this constant  $C$  to be a very large value. So let's imagine we set  $C$  to a very large value, may be a hundred thousand, some huge number. Let's see what the support vector machine will do.

### SVM Decision Boundary

$$\min_{\theta} C \sum_{i=1}^m [y^{(i)} \text{cost}_1(\theta^T x^{(i)}) + (1 - y^{(i)}) \text{cost}_0(\theta^T x^{(i)})] + \frac{1}{2} \sum_{j=1}^n \theta_j^2$$

Whenever  $y^{(i)} = 1$ :

$$\theta^T x^{(i)} \geq 1$$

$$\min_{\theta} C \cdot 0 + \frac{1}{2} \sum_{j=1}^n \theta_j^2$$

$$\text{s.t. } \theta^T x^{(i)} \geq 1 \quad \text{if } y^{(i)} = 1$$

$$\theta^T x^{(i)} \leq -1 \quad \text{if } y^{(i)} = 0$$

Whenever  $y^{(i)} = 0$ :

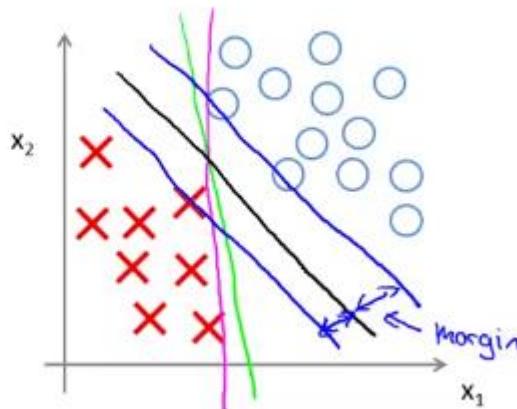
$$\theta^T x^{(i)} \leq -1$$

If  $C$  is very, very large, then when minimizing this optimization objective, we're going to be highly motivated to choose a value, so that this first term is equal to zero. So let's try to understand the optimization problem in the context of, what would it take to make this first term in the objective equal to zero, because you know, maybe we'll set  $C$  to some huge constant. And this will hope, this should give us additional intuition about what sort of hypotheses a support vector machine learns. So we saw already that whenever you have a training example with a

label of  $y=1$  if you want to make that first term zero, what you need is to find a value of  $\theta$  so that  $\theta^T x^{(i)} \geq 1$ .

And similarly, whenever we have an example, with label zero, in order to make sure that the cost,  $\text{Cost}_0(z)$ , in order to make sure that cost is zero we need that  $\theta^T x^{(i)}$  is less than or equal to -1. So, if we think of our optimization problem as now, really choosing parameters and show that this first term is equal to zero, what we're left with is the following optimization problem. We're going to minimize that first term zero, so  $C$  times zero, because we're going to choose parameters so that's equal to zero, plus one half and then you know that second term and this first term is ' $C$ ' times zero, so let's just cross that out because I know that's going to be zero. And this will be subject to the constraint that  $\theta^T x^{(i)}$  is greater than or equal to one, if  $y^{(i)} = 1$  and  $\theta^T x^{(i)}$  is less than or equal to minus one whenever you have a negative example. And it turns out that when you solve this optimization problem, when you minimize this as a function of the parameters  $\theta$  you get a very interesting decision boundary.

### SVM Decision Boundary: Linearly separable case



### Large margin classifier

Concretely, if you look at a data set like this with positive and negative examples, this data is linearly separable and by that, I mean that there exists, you know, a straight line, although there is many a different straight lines, they can separate the positive and negative examples perfectly.

For example, here is one decision boundary that separates the positive and negative examples, but somehow that doesn't look like a very natural one, right? Or by drawing an even worse one, you know here's another decision boundary that separates the positive and negative examples but just barely. But neither of those seem like particularly good choices. The Support Vector Machines will instead choose this decision boundary, which I'm drawing in black. And that seems like a much better decision boundary than either of the ones that I drew in magenta or in green. The black line seems like a more robust separator that you know it does a better job of separating the positive and negative examples.

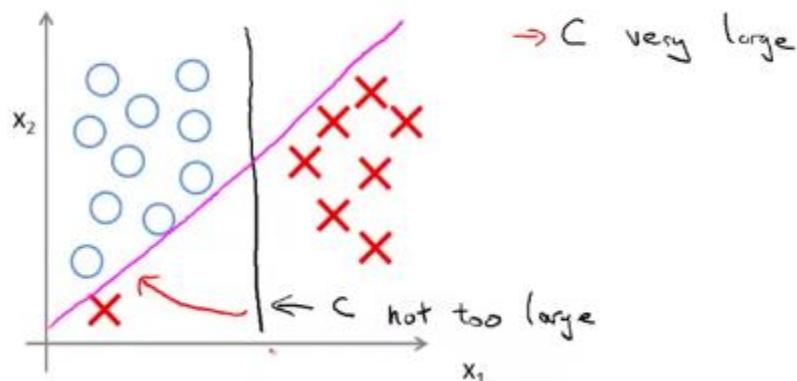
And mathematically, what that does is, this black decision boundary has a larger distance. That distance is called the margin, when I draw up this two extra blue lines, we see that the black decision boundary has some larger minimum distance from any of my training examples, whereas the magenta and the green lines they come awfully close to the training examples. And

then that seems to do a less a good job separating the positive and negative classes than my black line.

And so this distance is called the margin of the support vector machine and this gives the SVM a certain robustness, because it tries to separate the data with as a large a margin as possible. So the support vector machine is sometimes also called a large margin classifier and this is actually a consequence of the optimization problem we wrote down on the previous slide. I know that you might be wondering how is it that the optimization problem I wrote down in the previous slide, how does that lead to this large margin classifier.

I know I haven't explained that yet. And in the next video I'm going to sketch a little bit of the intuition about why that optimization problem gives us this large margin classifier. But this is a useful feature to keep in mind if you are trying to understand what are the sorts of hypothesis that an SVM will choose. That is, trying to separate the positive and negative examples with as big a margin as possible.

### Large margin classifier in presence of outliers



I want to say one last thing about large margin classifiers in this intuition, so we wrote out this large margin classification setting in the case of when C, that regularization concept, was very large, I think I set that to a hundred thousand or something. So given a dataset like this, maybe we'll choose that decision boundary that separates the positive and negative examples on large margin.

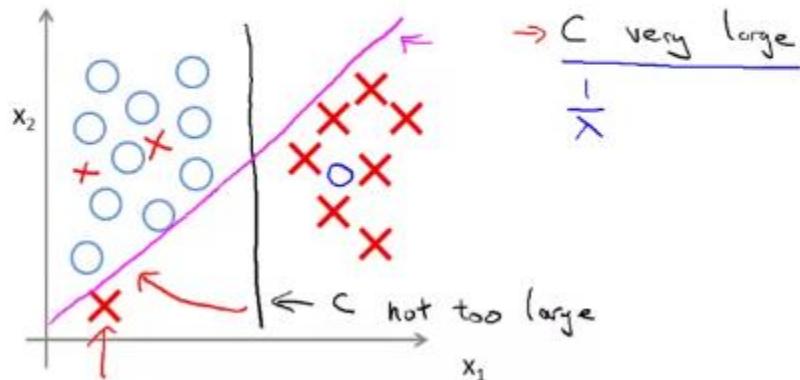
Now, the SVM is actually slightly more sophisticated than this large margin view might suggest. And in particular, if all you're doing is use a large margin classifier then your learning algorithms can be sensitive to outliers, so let's just add an extra positive example like that shown on the screen.

If we had one example then it seems as if to separate data with a large margin, maybe I'll end up learning a decision boundary like that, right? that is the magenta line and it's really not clear that based on the single outlier based on a single example and it's really not clear that it's actually a good idea to change my decision boundary from the black one over to the magenta one.

So, if C, if the regularization parameter C were very large, then this is actually what SVM will do, it will change the decision boundary from the black to the magenta one but if C were

reasonably small if you were to use the C, not too large then you still end up with this black decision boundary.

### Large margin classifier in presence of outliers



And of course if the data were not linearly separable so if you had some positive examples in here, or if you had some negative examples in here then the SVM will also do the right thing.

And so this picture of a large margin classifier that's really, that's really the picture that gives better intuition only for the case of when the regularization parameter C is very large, and just to remind you this corresponds C plays a role similar to one over Lambda, where Lambda is the regularization parameter we had previously. And so it's only of one over Lambda is very large or equivalently if Lambda is very small that you end up with things like this Magenta decision boundary, but in practice when applying support vector machines, when C is not very very large like that, it can do a better job ignoring the few outliers like here. And also do fine and do reasonable things even if your data is not linearly separable.

But when we talk about bias and variance in the context of support vector machines which will do a little bit later, hopefully all of these trade-offs involving the regularization parameter will become clearer at that time.

So I hope that gives some intuition about how this support vector machine functions as a large margin classifier that tries to separate the data with a large margin, technically this picture of this view is true only when the parameter C is very large, which is a useful way to think about support vector machines.

There was one missing step in this video which is, why is it that the optimization problem we wrote down on these slides, how does that actually lead to the large margin classifier, I didn't do that in this video, in the next video I will sketch a little bit more of the math behind that to explain that separate reasoning of how the optimization problem we wrote out results in a large margin classifier.

#### Mathematics behind Large Margin Classification

In this video, I'd like to tell you a bit about the math behind large margin classification. This video is optional, so please feel free to skip it. It may also give you better intuition about how the optimization problem of the support vector machine, how that leads to large margin classifiers.

==

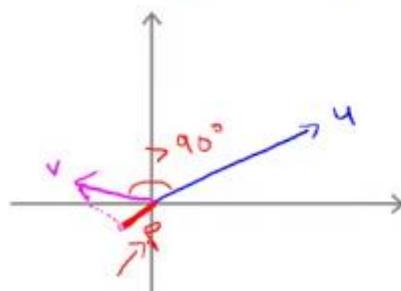
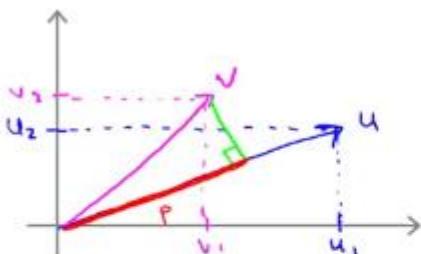
$$\rightarrow u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \quad \rightarrow v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

$$u^T v = ?$$

In order to get started, let me first remind you of a couple of properties of what vector inner products look like. Let's say I have two vectors U and V that look like this. So both two dimensional vectors. Then let's see what  $U^T V$  looks like. And **U transpose V** is also called the **inner product** between the vectors U and V.

Use a two dimensional vector, so I can plot it on this figure.

### Vector Inner Product



$$\rightarrow u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \quad \rightarrow v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

$$u^T v = ? \quad [u_1 \ u_2] \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

$$\|u\| = \text{length of vector } u \\ = \sqrt{u_1^2 + u_2^2} \in \mathbb{R}$$

$$p = \text{length of projection of } v \text{ onto } u. \\ u^T v = p \cdot \|u\| \leftarrow = v^T u \\ \text{Signed} \quad = u_1 v_1 + u_2 v_2 \leftarrow p \in \mathbb{R}$$

$$u^T v = p \cdot \|u\|$$

$$p < 0$$

Andrew N

So let's say that's the vector U. And what I mean by that is if on the horizontal axis that value takes whatever value  $U_1$  is and on the vertical axis the height of that is whatever  $U_2$  is the second component of the vector U. Now, one quantity that will be nice to have is the **norm of the vector U**. So, these are, you know, double bars on the left and right that denotes the norm or length of U. So this just means; really the Euclidean length of the vector U. And this is Pythagoras theorem is just equal to  $U_1$  squared plus  $U_2$  squared square root, right? And this is the length of the vector U. That's a real number. Just say you know, what is the length of this, what is the length of this vector down here, what is the length of this arrow that I just drew, is the normal view,

Now let's go back and look at the vector V because **we want to compute the inner product**. So V will be some other vector with, you know, some value  $V_1, V_2$ . And so, the vector V will look like that, towards V like so. Now let's go back and look at how to compute the inner product between U and V. **Here's how you can do it**. Let me take the vector V and project it down onto the vector U. So I'm going to take a orthogonal projection or a 90 degree projection, and project it down onto U like so. And what I'm going to do is measure length of this red line that I just drew here. So, I'm going to call the length of that **red line P**. So, P is the length or is the

magnitude of the projection of the vector  $V$  onto the vector  $U$ . Let me just write that down. So,  $P$  is the length of the projection of the vector  $V$  onto the vector  $U$ . And it is possible to show that **inner product  $U$  transpose  $V$** , that this is going to be equal to  **$P$  times the norm or the length of the vector  $U$** . So, this is one way to compute the inner product.

And if you actually do the geometry and figure out what  $P$  is and figure out what the norm of  $U$  is, this should give you the same way, the same answer as the other way of computing inner product, right, which is if you take  $U$  transpose  $V$  then  $U$  transposes this  $U_1 U_2$ , its a one by two matrix, 1 times  $V$ . And so this should actually give you  $U_1 V_1 + U_2 V_2$ . And so the theorem of linear algebra that **these two formulas give you the same answer**.

And by the way,  **$U$  transpose  $V$  is also equal to  $V$  transpose  $U$** . So if you were to do the same process in reverse, instead of projecting  $V$  onto  $U$ , you could project  $U$  onto  $V$ . Then, you know, do the same process, but with the rows of  $U$  and  $V$  reversed. And you would actually, you should actually get the same number whatever that number is.

And just to clarify what's going on in this equation the norm of  $U$  is a real number and  $P$  is also a real number. And so  $U$  transpose  $V$  is the regular multiplication of two real numbers, of the length of  **$P$  times the norm of  $U$** .

Just one last detail, which is if you look at the norm of  $P$ ,  $P$  is actually signed so to the right. And it can either be positive or negative. So let me say what I mean by that. If  $U$  is a vector that looks like this and  $V$  is a vector that looks like this. So if the angle between  $U$  and  $V$  is greater than ninety degrees. Then if I project  $V$  onto  $U$ , what I get is a projection it looks like this and so that length  $P$ . And in this case, I will still have that  $U$  transpose  $V$  is equal to  $P$  times the norm of  $U$ . Except in this example  $P$  will be negative. So, you know, in inner products if the angle between  $U$  and  $V$  is less than ninety degrees, then  $P$  is the positive length for that red line whereas if the angle of this angle of here is greater than 90 degrees then  $P$  here will be negative of the length of the line, of that little line segment right over there.

So the inner product between two vectors can also be negative if the angle between them is greater than 90 degrees. So that's how vector inner products work.

We're going to use these properties of vector inner product to try to understand the support vector machine optimization objective over there.

$\omega = (\sum \theta_j)^2$

**SVM Decision Boundary**

$$\min_{\theta} \frac{1}{2} \sum_{j=1}^n \theta_j^2 = \frac{1}{2} (\theta_1^2 + \theta_2^2) = \frac{1}{2} (\underbrace{\theta_1^2 + \theta_2^2}_{= \|\theta\|^2}) = \frac{1}{2} \|\theta\|^2$$

s.t.  $\theta^T x^{(i)} \geq 1 \quad \text{if } y^{(i)} = 1$   
 $\theta^T x^{(i)} \leq -1 \quad \text{if } y^{(i)} = 0$   
Simplification:  $\theta_0 = 0$ .  $n=2$

$$\begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix} \quad \theta_0 = 0$$

Here is the optimization objective for the support vector machine that we worked out earlier. Just for the purpose of this slide I am going to make one simplification or one just to make the objective easy to analyze. And what I'm going to do is ignore the intercept terms. So, we'll just ignore  $\theta_0$  and set that to be equal to 0. To make things easier to plot, I'm also going to set  $n$  the number of features to be equal to 2. So, we have only 2 features,  $X_1$  and  $X_2$ .

Now, let's look at the objective function. The optimization objective of the SVM. When we have only two features, when  $n = 2$ , this can be written,  $\frac{1}{2}(\theta_1^2 + \theta_2^2)$  because we only have two parameters,  $\theta_1$  and  $\theta_2$ . And what I'm going to do is to rewrite this a bit. I'm going to write this as one half of theta one squared plus theta two squared and the square root squared. And the reason I can do that, is because for any number, you know,  $W$ , right, the square roots of  $W$  and then squared, that's just equal to  $W$ . So square roots and squared should give you the same thing.

What you may notice is that this term inside the parentheses is that's equal to the norm or the length of the vector  $\theta$ , and what I mean by that is that if we write out the vector  $\theta$  like this, as you know  $\theta_1, \theta_2$ . Then this term that I've just underlined in red, that's exactly the length, or the norm, of the vector  $\theta$ . We are calling the definition of the norm of the vector that we had on the previous slide. And in fact this is actually equal to the length of the vector  $\theta$ , whether you write it as  $\theta_0, \theta_1, \theta_2$ . That is, if  $\theta_0$  is equal to zero, as I assume here. Or just the length of  $\theta_1, \theta_2$ ; but for this line I am going to ignore  $\theta_0$ . So let me just, you know, treat  $\theta$  as this, let me just write  $\theta$ , the normal  $\theta$  as this  $\theta_1, \theta_2$  only, but the math works out either way, whether we include theta zero here or not. So it's not going to matter for the rest of our derivation.

And so finally this means that my optimization objective is equal to one half of the norm of theta squared. So all the support vector machine is doing, in the optimization objective, is it's minimizing the squared norm of the square length of the parameter vector  $\theta$ .

Now what I'd like to do is look at these terms,  $\theta^T x$  and understand better what they're doing.

So given the parameter vector  $\theta$  and given example  $x$ , what is this equal to? And on the previous slide, we figured out what  $U$  transpose  $V$  looks like, with different vectors  $U$  and  $V$ , and so we're going to take those definitions, you know, with  $\theta$  and  $x^{(i)}$  playing the roles of  $U$  and  $V$ . And let's see what that picture looks like. So, let's say I plot. Let's say I look at just a single training example. Let's say I have a positive example, the drawing was a cross there and let's say that is my example  $x^{(i)}$ , so what that really means is that I plotted on the horizontal axis some value  $x_1^{(i)}$  and on the vertical axis  $x_2^{(i)}$ . That's how I plot my training examples.

And although we haven't been really thinking of this as a vector, what this really is, this is a vector from the origin from 0, 0 out to the location of this training example. And now let's say we have a parameter vector and I'm going to plot that as vector, as well. What I mean by that is if I plot  $\theta_1$  here and  $\theta_2$  there, so what is the inner product  $\theta^T x^{(i)}$ .

While using our earlier method, the way we compute that is we take my example and project it onto my parameter vector  $\theta$ . And then I'm going to look at the length of this segment that I'm coloring in, in red. And I'm going to call that  $P^{(i)}$  to denote that this is a projection of the  $i$ -th training example onto the parameter vector  $\theta$ .

And so what we have is that  $\theta^T x^{(i)}$  is equal to following what we have on the previous slide, this is going to be equal to  $P$  times the length of the norm of the vector  $\theta$ . And this is of course also equal to  $\theta_1 x_1^{(i)}$  plus  $\theta_2 x_2^{(i)}$ . So each of these is, you know, an equally valid way of computing the inner product between  $\theta$  and  $x^{(i)}$ .

Okay. So where does this leave us? What this means is that, these constrains that  $\theta^T x^{(i)}$  be greater than or equal to one or less than minus one. What this means is that it can replace these constraints that  $P^{(i)}$  times  $X$  be greater than or equal to one. Because  $\theta^T x^{(i)}$  is equal to  $P^{(i)}$  times the norm of theta.

### SVM Decision Boundary

$$\min_{\theta} \frac{1}{2} \sum_{j=1}^n \theta_j^2 = \frac{1}{2} \|\theta\|^2$$

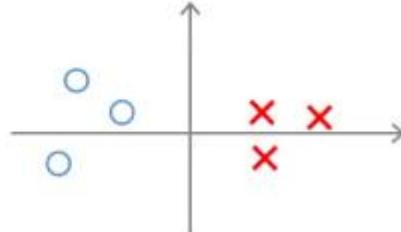
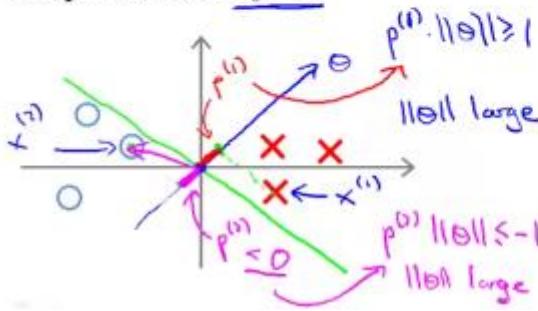
s.t.

$$P^{(i)} \cdot \|\theta\| \geq 1 \quad \text{if } y^{(i)} = 1$$

$$P^{(i)} \cdot \|\theta\| \leq -1 \quad \text{if } y^{(i)} = -1$$

where  $P^{(i)}$  is the projection of  $x^{(i)}$  onto the vector  $\theta$ .

Simplification:  $\theta_0 = 0$



So writing that into our optimization objective, this is what we get where I have instead of  $\theta^T x^{(i)}$ , I now have this  $P^{(i)}$  times the norm of  $\theta$ . And just to remind you we worked out earlier too that this optimization objective can be written as one half times the norm of  $\theta$  squared. So, now let's consider the training example that we have at the bottom and for now, continuing to use the simplification that  $\theta_0 = 0$ . Let's see what decision boundary the support vector machine will choose.

Here's one option, let's say the support vector machine were to choose this decision boundary. This is not a very good choice because it has very small margins. This decision boundary comes very close to the training examples. Let's see why the support vector machine will not do this. For this choice of parameters it's possible to show that the parameter vector  $\theta$  is actually at 90 degrees to the decision boundary. And so, that green decision boundary corresponds to a parameter vector  $\theta$  that points in that direction. And by the way, the simplification that  $\theta_0 = 0$  that just means that the decision boundary has to pass through the origin (0,0) over there.

So now, let's look at what this implies for the optimization objective. Let's say that this example here. Let's say that's my first example, you know,  $x^{(1)}$ . If we look at the projection of this example onto my parameters  $\theta$ , that's the projection, and so that little red line segment, that is equal to  $P^{(1)}$ . And that is going to be pretty small, right. And similarly, if this example here, if this happens to be  $x^{(2)}$ , that's my second example. Then, if I look at the projection of this example onto  $\theta$ , you know, then, let me draw this one in magenta. This little magenta line segment, that's going to be  $P^{(2)}$ . That's the projection of the second example onto my, onto the direction of my parameter vector  $\theta$ , which goes like this. And so, this little projection line segment is getting pretty small.  $P^{(2)}$  will actually be a negative number, right so  $P^{(2)}$  is in the opposite direction.

This vector has greater than 90 degree angle with my parameter vector  $\theta$ , it's going to be less than 0. And so what we're finding is that these terms  $P^{(i)}$  are going to be pretty small numbers. So if we look at the optimization objective and see, well, for positive examples we need  $P^{(i)}$  times the norm of  $\theta$  to be bigger than or equal to one. But if  $P^{(i)}$  over here, if  $P^{(1)}$  over here is pretty small, that means that we need the norm of  $\theta$  to be pretty large, right? Because if  $P^{(1)}$  of  $\theta$  is small and we want  $P^{(1)}$  you know times in all of  $\theta$  to be bigger than either one, well the only way for that to be true for the product of these two numbers to be large if  $P^{(1)}$  is small, as we said, we want the norm of  $\theta$  to be large.

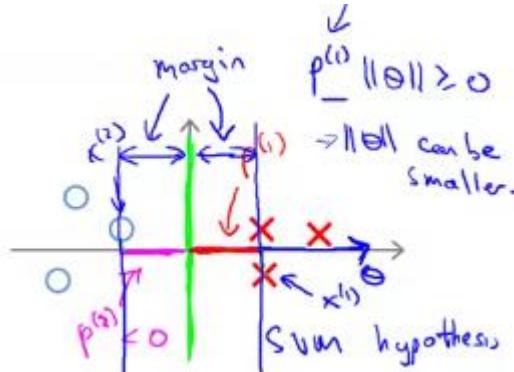
And similarly for our negative example, we need  $P^{(2)}$  times the norm of  $\theta$  to be less than or equal to minus one. And we saw in this example, right, that  $P_2$  is going to be a pretty small negative number, and so the only way for that to happen as well is for the norm of  $\theta$  to be large.

But what we are doing in the optimization objective is we are trying to find a setting of parameters where the norm of  $\theta$  is small, and so you know, so this doesn't seem like such a good direction for the parameter vector  $\theta$ .

In contrast, just look at a different decision boundary. Here, let's say, this SVM chooses that decision boundary.

Now the picture is going to be very different. If that is the decision boundary, here is the corresponding direction for  $\theta$ . So, with the decision boundary you know, that vertical line that corresponds to it, it is possible to show using linear algebra that the way to get that green decision boundary is to have the vector  $\theta$  to be at 90 degrees to it.

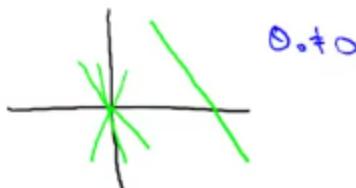
And now if you look at the projection of your data onto the vector  $x$ . Lets say that it's before this example is my example of  $x^{(1)}$ .



So when I project this on to  $x$ , or onto  $\theta$ , what I find is that this is  $P^{(1)}$ , that length there is  $P^{(1)}$ . And some other example, that example is  $x^{(2)}$  and I do the same projection and what I find is that this length here is a  $P^{(2)}$ , really that is going to be less than 0. And you notice that now  $P^{(1)}$  and  $P^{(2)}$ , these lengths of the projections are going to be much bigger. And so if we still need to enforce these constraints that  $P^{(1)} \|\theta\| \geq 1$  because  $P_1$  is so much bigger now the norm of  $\theta$  can be smaller.

And so, what this means is that by choosing the decision boundary shown on the right instead of on the left, the SVM can make the norm of the parameters  $\theta$  much smaller. So, if we can make the norm of  $\theta$  smaller and therefore make the squared norm of  $\theta$  smaller, which is why the SVM would choose this hypothesis on the right instead. And this is how the SVM gives rise to this large margin classification effect.

Mainly, if you look at this green line, if you look at this green hypothesis we want the projections of my positive and negative examples onto theta to be large, and the only way for that to hold true this is if surrounding this green line, there's this large margin, there's this large gap that separates the positive and negative examples and is really the magnitude of this gap. The magnitude of this margin is exactly the values of  $P^{(1)}$ ,  $P^{(2)}$ ,  $P^{(3)}$  and so on. And so as by making the margin large, by making these terms  $P^{(1)}$ ,  $P^{(2)}$ ,  $P^{(3)}$  and so on that's the **SVM can end up with a smaller value for the norm of  $\theta$**  which is what it is trying to do in the objective. And this is why this support vector machine ends up with a large margin classifier, because it's trying to maximize the norm of these  $P^{(i)}$ , which is the distance from the training examples to the decision boundary.



Finally, we did this whole derivation using this simplification that the parameter  $\theta_0$  must be equal to 0. The effect of that as I mentioned briefly, is that if  $\theta_0$  is equal to 0 what that means is that we are entertaining decision boundaries that pass through the origins of decision boundaries pass through the origin like that. If you allow  $\theta_0$  to be non 0 then what that means is that you entertain the decision boundaries that did not cross through the origin, like that one I just drew. And I'm not going to do the full derivation that, but it turns out that this same large margin proof works in pretty much in exactly the same way. And there's a generalization of this argument that we just went through and not long ago through that shows that even when  $\theta_0$  is non 0, what the SVM is trying to do when you have this optimization objective, which again corresponds to the case of when C is very large. But it is possible to show that, you know, when  $\theta_0$  is not equal to 0 this support vector machine is still finding, is really trying to find the large margin separator between the positive and negative examples. So that explains how this support vector machine is a large margin classifier.

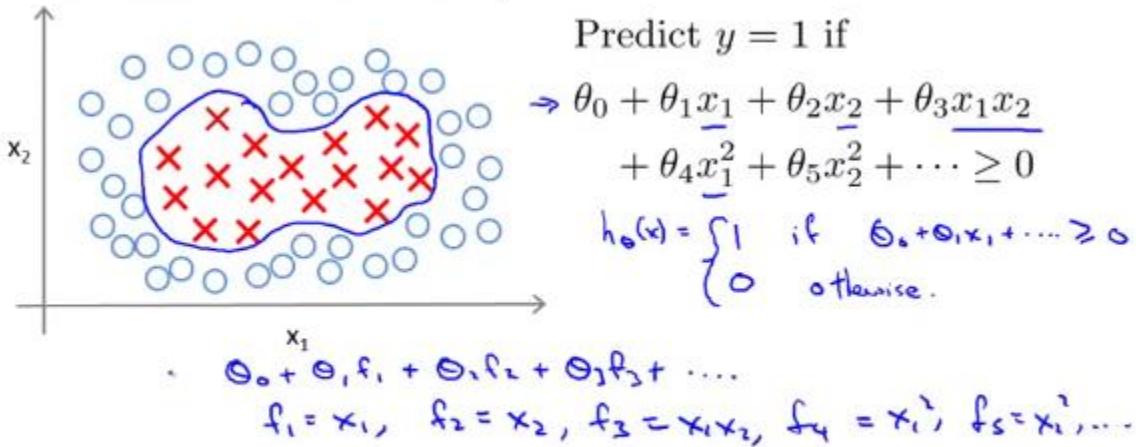
In the next video we will start to talk about how to take some of these SVM ideas and start to apply them to build a complex nonlinear classifiers.

### Kernels I

In this video, I'd like to start adapting support vector machines in order to develop complex nonlinear classifiers.

The main technique for doing that is something called kernels. Let's see what this kernels are and how to use them.

### Non-linear Decision Boundary



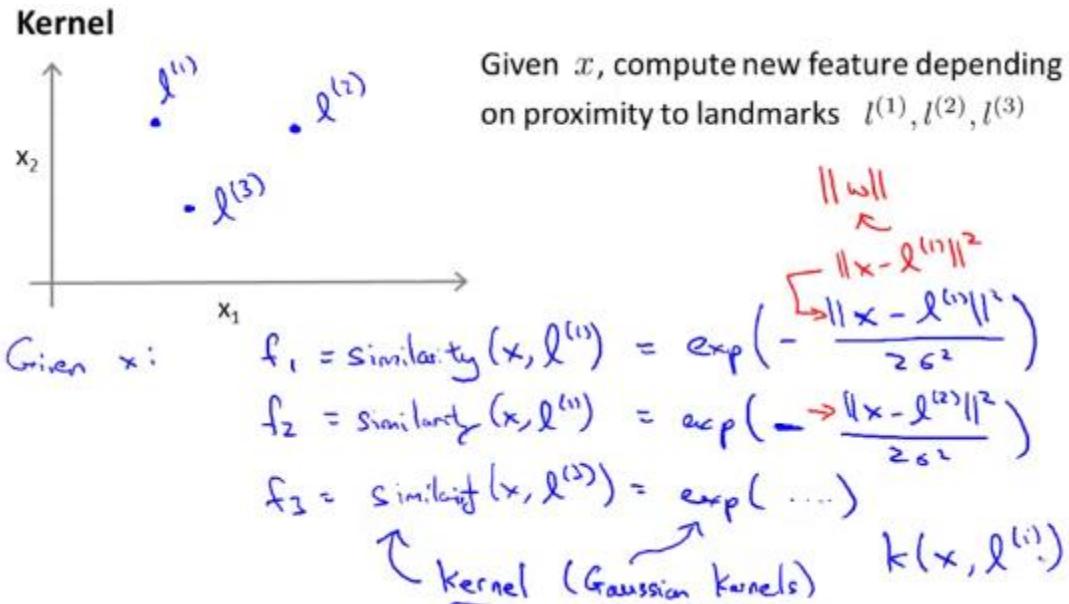
Is there a different / better choice of the features  $f_1, f_2, f_3, \dots$ ?

If you have a training set that looks like this, and you want to find a nonlinear decision boundary to distinguish the positive and negative examples, maybe a decision boundary that looks like that.

One way to do so is to come up with a set of complex polynomial features, right? So, set of features that looks like this, so that you end up with a hypothesis X that predicts 1 if you know that  $\theta_0$  and plus  $\theta_1 x_1$  plus dot dot dot all those polynomial features is greater than 0, and predict

0, otherwise. And another way of writing this is to introduce a little bit of new notation that I'll use later, is that we can think of a hypothesis as computing a decision boundary using this. So,  $\theta_0$  plus  $\theta_1 f_1$  plus  $\theta_2 f_2$  plus  $\theta_3 f_3$  plus and so on. Where I'm going to use this new notation  $f_1, f_2, f_3$  and so on to denote these new sort of features that I'm computing, so  $f_1$  is just  $x_1$ ,  $f_2$  is equal to  $x_2$ ,  $f_3$  is equal to this one here so,  $x_1 x_2$ . So,  $f_4$  is equal to  $x_1^2$  where  $f_5$  is to be  $x_2^2$  and so on. And we have seen previously that coming up with these high order polynomials is one way to come up with lots more features. But the question is, is there a different choice of features or is there a better choice of features than these high order polynomials, because you know it's not clear that these high order polynomial is what we want, and what we talked about computer vision, talked about when the input is an image with lots of pixels. We also saw how using high order polynomials becomes very computationally expensive because there are a lot of these higher order polynomial terms.

So, is there a different or a better choice of the features that we can use to plug into this sort of hypothesis form?



So, here is one idea for how to define new features  $f_1, f_2, f_3$ . On this slide I am going to define only three new features, but for real problems we can get to define a much larger number. But here's what I'm going to do. In this place of features  $x_1, x_2$ , and I'm going to leave  $x_0$  out of this, the intercept term  $x_0$ , but in this space  $x_1 x_2$ , I'm going to just, you know, manually pick a few points, and then call these points  $l^{(1)}$ , we are going to pick a different point, let's call that  $l^{(2)}$  and let's pick the third one and call this one  $l^{(3)}$ , and for now let's just say that I'm going to choose these three points manually. I'm going to call these three points' landmarks, so landmark one, two, three.

What I'm going to do is define my new features as follows, given an example  $x$ , let me define my first feature  $f_1$  to be some measure of the similarity between my training example  $x$  and my first landmark  $l^{(1)}$  and this specific formula that I'm going to use to measure similarity is going to be, this is  $e$  to the minus the length of  $X$  minus  $l_1$ , squared, divided by two sigma squared.

So, depending on whether or not you watched the previous optional video, this notation, you know, this is the length of the vector  $W$ . And so, this thing here, this  $X$  minus  $l_1$ , this is actually just the Euclidean distance squared, is the Euclidean distance between the point  $x$  and the landmark  $l^{(1)}$ .

We will see more about this later. But that's my first feature, and my second feature  $f_2$  is going to be, you know, **similarity function** that measures how similar  $x$  is to  $l^{(2)}$  and again is going to be defined as the following function, this is  $e$  to the minus of the square of the Euclidean distance between  $x$  and the second landmark, that is what the enumerator is, and then divided by  $2\sigma^2$  squared and similarly  $f_3$  is, you know, similarity between  $x$  and  $l^{(3)}$ , which is equal to, again, similar formula.

And what this **similarity function** is, **the mathematical term for** this, is that this is going to be **a kernel function**.

And the specific kernel I'm using here, this is actually **called a Gaussian kernel**. And so this formula, this particular choice of similarity function is called a Gaussian kernel. But the way the terminology goes is that, you know, in the abstract these different similarity functions are called kernels and we can have different similarity functions and the specific example I'm giving here is called the Gaussian kernel.

We'll see other examples of other kernels. But for now just think of these as similarity functions. And so, instead of writing similarity between  $X$  and  $l$ , sometimes we also write this a kernel denoted you know, **lower case k between x and one of my landmarks** all right.

So **let's see what these kernels actually do** and why these sorts of similarity functions, why these expressions might make sense.

### Kernels and Similarity

$$f_1 = \text{similarity}(x, l^{(1)}) = \exp\left(-\frac{\|x - l^{(1)}\|^2}{2\sigma^2}\right) = \exp\left(-\frac{\sum_{j=1}^n (x_j - l_j^{(1)})^2}{2\sigma^2}\right)$$

If  $\underline{x} \approx l^{(1)}$  :

$$f_1 \approx \exp\left(-\frac{0^2}{2\sigma^2}\right) \approx 1$$

$$\begin{array}{ccc} l^{(1)} & \rightarrow & f_1 \\ l^{(2)} & \rightarrow & f_2 \\ l^{(3)} & \rightarrow & f_3 \end{array}$$

If  $\underline{x}$  if far from  $\underline{l}^{(1)}$  :

$$f_1 = \exp\left(-\frac{(\text{large number})^2}{2\sigma^2}\right) \approx 0.$$

$$\begin{array}{c} \uparrow \\ X \end{array}$$

So let's take my first landmark, my landmark  $l^{(1)}$ , which is one of those points I chose on my figure just now. So the similarity of the kernel between  $x$  and  $l^{(1)}$  is given by this expression. Just to make sure, you know, we are on the same page about what the numerator term is, the numerator can also be written as a sum from  $j$  equals 1 through  $n$  on sort of the distance. So this

is the component wise distance between the vector  $x$  and the vector  $l$ . And again for the purpose of these slides I'm ignoring  $x_0$ .

So just ignoring the intercept term  $x_0$ , which is always equal to 1. So, you know, this is how you compute the kernel with similarity between  $x$  and a landmark.

So let's see what this function does. Suppose  $x$  is close to one of the landmarks. Then this Euclidean distance formula in the numerator will be close to 0, right, that is this term here, the distance, square of the distance using  $x$  and  $l$  will be close to zero. And so  $f_1$ , this is a simple feature, will be approximately  $e$  to the minus 0 and then the numerator squared over 2 is equal to squared so that  $e$  to the 0,  $e$  to minus 0,  $e$  to 0 is going to be close to one.

And I'll put the approximation symbol here because you know the distance may not be exactly 0, but if  $x$  is closer to landmark this term will be close to 0 and so  $f_1$  would be close 1.

Conversely, if  $X$  is far from  $l^{(1)}$  then this first feature  $f_1$  will be  $e$  to the minus of some large number squared, divided by two sigma squared and  $e$  to the minus of a large number is going to be close to 0. So what these features do is they measure how similar  $x$  is from one of your landmarks and the feature  $f$  is going to be close to one when  $x$  is close to your landmark and is going to be 0 or close to zero when  $x$  is far from your landmark. Each of these landmarks, on the previous slide, I drew three landmarks,  $l^{(1)}, l^{(2)}, l^{(3)}$ , each of these landmarks, defines a new feature  $f_1, f_2$  and  $f_3$ . That is, given the training example  $x$ , we can now compute three new features:  $f_1, f_2$  and  $f_3$ , given, you know, the three landmarks that I wrote just now.

But first, let's look at this exponentiation function, let's look at this similarity function and plot in some figures and just, you know, understand better what this really looks like.

==

For this example, let's say I have two features  $X_1$  and  $X_2$ . And let's say my first landmark,  $l_1$  is at a location, 3 5. So and let's say I set sigma squared equals one for now. If I plot what this feature looks like, what I get is this figure. So the vertical axis, the height of the surface is the value of  $f_1$  and down here on the horizontal axis are, if I have some training example, and there is  $x_1$  and there is  $x_2$ .

Given a certain training example, the training example here which shows the value of  $x_1$  and  $x_2$  at a height above the surface, shows the corresponding value of  $f_1$  and down below this is the same figure I had showed, using a quantifiable plot, with  $x_1$  on horizontal axis,  $x_2$  on horizontal axis and so, this figure on the bottom is just a contour plot of the 3D surface. You notice that when  $X$  is equal to 3 5 exactly, then we the  $f_1$  takes on the value 1, because that's at the maximum and  $X$  moves away as  $X$  goes further away then this feature takes on values that are close to 0.

And so, this is really a feature,  $f_1$  measures, you know, how close  $X$  is to the first landmark and it varies between 0 and one depending on how close  $X$  is to the first landmark  $l_1$ . Now the other was due on this slide is show the effects of varying this parameter sigma squared. So, sigma squared is the parameter of the Gaussian kernel and as you vary it, you get slightly different effects. Let's set sigma squared to be equal to 0.5 and see what we get. We set sigma square to

0.5, what you find is that the kernel looks similar, except for the width of the bump becomes narrower. The contours shrink a bit too. So if sigma squared equals to 0.5 then as you start from X equals 3 5 and as you move away, then the feature f1 falls to zero much more rapidly and conversely, if you have increase since where three in that case and as I move away from, you know 1. So this point here is really 1, right, that's l1 is at location 3 5, right. So it's shown up here.

And if sigma squared is large, then as you move away from l1, the value of the feature falls away much more slowly. So, given this definition of the features, let's see what source of hypothesis we can learn. Given the training example X, we are going to compute these features f1, f2, f3 and a hypothesis is going to predict one when theta 0 plus theta 1 f1 plus theta 2 f2, and so on is greater than or equal to 0.

For this particular example, let's say that I've already found a learning algorithm and let's say that, you know, somehow I ended up with these values of the parameter. So if theta 0 equals minus 0.5, theta 1 equals 1, theta 2 equals 1, and theta 3 equals 0. And what I want to do is consider what happens if we have a training example that takes has location at this magenta dot, right where I just drew this dot over here. So let's say I have a training example X, what would my hypothesis predict?

Well, If I look at this formula. Because my training example X is close to l1, we have that f1 is going to be close to 1 the because my training example X is far from l2 and l3 I have that, you know, f2 would be close to 0 and f3 will be close to 0. So, if I look at that formula, I have theta 0 plus theta 1 times 1 plus theta 2 times some value. Not exactly 0, but let's say close to 0. Then plus theta 3 times something close to 0. And this is going to be equal to plugging in these values now. So, that gives minus 0.5 plus 1 times 1 which is 1, and so on. Which is equal to 0.5 which is greater than or equal to 0. So, at this point, we're going to predict Y equals 1, because that's greater than or equal to zero. Now let's take a different point. Now lets' say I take a different point, I'm going to draw this one in a different color, in cyan say, for a point out there, if that were my training example X, then if you make a similar computation, you find that f1, f2, f3 are all going to be close to 0. And so, we have theta 0 plus theta 1, f1, plus so on and this will be about equal to minus 0.5, because theta 0 is minus 0.5 and f1, f2, f3 are all zero. So this will be minus 0.5, this is less than zero. And so, at this point out there, we're going to predict Y equals zero. And if you do this yourself for a range of different points, be sure to convince yourself that if you have a training example that's close to L2, say, then at this point we'll also predict Y equals one.

And in fact, what you end up doing is, you know, if you look around this boundary, this space, what we'll find is that for points near l1 and l2 we end up predicting positive. And for points far away from l1 and l2, that's for points far away from these two landmarks, we end up predicting that the class is equal to 0. As so, what we end up doing, is that the decision boundary of this hypothesis would end up looking something like this where inside this red decision boundary would predict Y equals 1 and outside we predict Y equals 0. And so this is how with this definition of the landmarks and of the kernel function. We can learn pretty complex non-linear decision boundary, like what I just drew where we predict positive when we're close to either one of the two landmarks.

And we predict negative when we're very far away from any of the landmarks. And so this is part of the idea of kernels of and how we use them with the support vector machine, which is that we define these extra features using landmarks and similarity functions to learn more complex nonlinear classifiers. So hopefully that gives you a sense of the idea of kernels and how we could use it to define new features for the Support Vector Machine. But there are a couple of questions that we haven't answered yet. One is, how do we get these landmarks? How do we choose these landmarks? And another is, what other similarity functions, if any, can we use other than the one we talked about, which is called the Gaussian kernel.

In the next video we give answers to these questions and put everything together to show how support vector machines with kernels can be a powerful way to learn complex nonlinear functions.

### Kernels II

In the last video, we started to talk about the kernels idea and how it can be used to define new features for the support vector machine.

In this video, I'd like to throw in some of the missing details and, also, say a few words about how to use these ideas in practice. Such as, how they pertain to, for example, the bias variance trade-off in support vector machines. In the last video, I talked about the process of picking a few landmarks.

You know,  $l_1$ ,  $l_2$ ,  $l_3$  and that allowed us to define the similarity function also called the kernel or in this example if you have this similarity function this is a Gaussian kernel. And that allowed us to build this form of a hypothesis function. But where do we get these landmarks from? Where do we get  $l_1$ ,  $l_2$ ,  $l_3$  from? And it seems, also, that for complex learning problems, maybe we want a lot more landmarks than just three of them that we might choose by hand. So in practice this is how the landmarks are chosen which is that given the machine learning problem. We have some data set of some some positive and negative examples. So, this is the idea here which is that we're gonna take the examples and for every training example that we have, we are just going to call it. We're just going to put landmarks as exactly the same locations as the training examples.

So if I have one training example if that is  $x_1$ , well then I'm going to choose this is my first landmark to be at exactly the same location as my first training example. And if I have a different training example  $x_2$ . Well we're going to set the second landmark to be the location of my second training example. On the figure on the right, I used red and blue dots just as illustration, the color of this figure, the color of the dots on the figure on the right is not significant. But what I'm going to end up with using this method is I'm going to end up with  $m$  landmarks of  $l_1$ ,  $l_2$  down to  $l_m$  if I have  $m$  training examples with one landmark per location of my per location of each of my training examples. And this is nice because it is saying that my features are basically going to measure how close an example is to one of the things I saw in my training set.

So, just to write this outline a little more concretely, given  $m$  training examples, I'm going to choose the location of my landmarks to be exactly near the locations of my  $m$  training examples. When you are given example  $x$ , and in this example  $x$  can be something in the training set, it can be something in the cross validation set, or it can be something in the test set.

Given an example  $x$  we are going to compute, you know, these features as so  $f_1, f_2$ , and so on. Where  $l_1$  is actually equal to  $x_1$  and so on. And these then give me a feature vector. So let me write  $f$  as the feature vector. I'm going to take these  $f_1, f_2$  and so on, and just group them into feature vector. Take those down to  $f_m$ . And, you know, just by convention. If we want, we can add an extra feature  $f_0$ , which is always equal to 1. So this plays a role similar to what we had previously. For  $x_0$ , which was our interceptor. So, for example, if we have a training example  $x(i), y(i)$ , the features we would compute for this training example will be as follows: given  $x(i)$ , we will then map it to, you know,  $f_1(i)$ . Which is the similarity. I'm going to abbreviate as SIM instead of writing out the whole word similarity, right? And  $f_2(i)$  equals the similarity between  $x(i)$  and  $l_2$ , and so on, down to  $f_m(i)$  equals the similarity between  $x(i)$  and  $l(m)$ . And somewhere in the middle. Somewhere in this list, you know, at the  $i$ -th component, I will actually have one feature component which is  $f$  subscript  $i(i)$ , which is going to be the similarity between  $x$  and  $l(i)$ . Where  $l(i)$  is equal to  $x(i)$ , and so you know  $f_i(i)$  is just going to be the similarity between  $x$  and itself. And if you're using the Gaussian kernel this is actually  $e$  to the minus 0 over 2 sigma squared and so, this will be equal to 1 and that's okay.

So one of my features for this training example is going to be equal to 1. And then similar to what I have above. I can take all of these  $m$  features and group them into a feature vector. So instead of representing my example, using, you know,  $x(i)$  which is this what  $R(n)$  plus  $R(n)$  one dimensional vector. Depending on whether you can set terms, is either  $R(n)$  or  $R(n)$  plus 1. We can now instead represent my training example using this feature vector  $f$ . I am going to write this  $f$  superscript  $i$ . Which is going to be taking all of these things and stacking them into a vector. So,  $f_1(i)$  down to  $f_m(i)$  and if you want and well, usually we'll also add this  $f_0(i)$ , where  $f_0(i)$  is equal to 1. And so this vector here gives me my new feature vector with which to represent my training example. So given these kernels and similarity functions, here's how we use a simple vector machine. If you already have a learning set of parameters  $\theta$ , then if you given a value of  $x$  and you want to make a prediction. What we do is we compute the features  $f$ , which is now an  $R(m)$  plus 1 dimensional feature vector. And we have  $m$  here because we have  $m$  training examples and thus  $m$  landmarks and what we do is we predict 1 if  $\theta^T f$  is greater than or equal to 0. Right. So, if  $\theta^T f$ , of course, that's just equal to  $\theta_0 + \theta_1 f_1 + \dots + \theta_m f_m$ . And so my parameter vector  $\theta$  is also now going to be an  $m$  plus 1 dimensional vector. And we have  $m$  here because where the number of landmarks is equal to the training set size. So  $m$  was the training set size and now, the parameter vector  $\theta$  is going to be  $m$  plus one dimensional.

So that's how you make a prediction if you already have a setting for the parameter's  $\theta$ . How do you get the parameter's  $\theta$ ? Well you do that using the SVM learning algorithm, and specifically what you do is you would solve this minimization problem. You've minimized the parameter's  $\theta$  of  $C$  times this cost function which we had before. Only now, instead of looking there instead of making predictions using  $\theta^T x(i)$  using our original features,  $x(i)$ . Instead we've taken the features  $x(i)$  and replace them with a new features so we are using  $\theta^T f(i)$  to make a prediction on the  $i$ 's training examples and we see that, you know, in both places here and it's by solving this minimization problem that you get the parameters for your Support Vector Machine. And one last detail is because this optimization problem we really have  $n$  equals  $m$  features. That is here. The number of features we have.

Really, the effective number of features we have is dimension of  $f$ . So that  $n$  is actually going to be equal to  $m$ . So, if you want to, you can think of this as a sum, this really is a sum from  $j$  equals 1 through  $m$ . And then one way to think about this, is you can think of it as  $n$  being equal to  $m$ , because if  $f$  isn't a new feature, then we have  $m$  plus 1 features, with the plus 1 coming from the interceptor.

And here, we still do sum from  $j$  equal 1 through  $n$ , because similar to our earlier videos on regularization, we still do not regularize the parameter theta zero, which is why this is a sum for  $j$  equals 1 through  $m$  instead of  $j$  equals zero though  $m$ . So that's the support vector machine learning algorithm. That's one sort of, mathematical detail aside that I should mention, which is that in the way the support vector machine is implemented, this last term is actually done a little bit differently.

So you don't really need to know about this last detail in order to use support vector machines, and in fact the equations that are written down here should give you all the intuitions that should need. But in the way the support vector machine is implemented, you know, that term, the sum of  $j$  of theta  $j$  squared right? Another way to write this is this can be written as theta transpose theta if we ignore the parameter theta 0. So theta 1 down to theta  $m$ . Ignoring theta 0. Then this sum of  $j$  of theta  $j$  squared that this can also be written theta transpose theta. And what most support vector machine implementations do is actually replace this theta transpose theta, will instead, theta transpose times some matrix inside, that depends on the kernel you use, times theta.

And so this gives us a slightly different distance metric. We'll use a slightly different measure instead of minimizing exactly the norm of theta squared means that minimize something slightly similar to it. That's like a rescale version of the parameter vector theta that depends on the kernel. But this is kind of a mathematical detail. That allows the support vector machine software to run much more efficiently. And the reason the support vector machine does this is with this modification. It allows it to scale to much bigger training sets. Because for example, if you have a training set with 10,000 training examples. Then, you know, the way we define landmarks, we end up with 10,000 landmarks. And so theta becomes 10,000 dimensional. And maybe that works, but when  $m$  becomes really, really big then solving for all of these parameters, you know, if  $m$  were 50,000 or a 100,000 then solving for all of these parameters can become expensive for the support vector machine optimization software, thus solving the minimization problem that I drew here.

So kind of as mathematical detail, which again you really don't need to know about. It actually modifies that last term a little bit to optimize something slightly different than just minimizing the norm squared of theta squared, of theta. But if you want, you can feel free to think of this as an kind of a n implementational detail that does change the objective a bit, but is done primarily for reasons of computational efficiency, so usually you don't really have to worry about this.

And by the way, in case your wondering why we don't apply the kernel's idea to other algorithms as well like logistic regression, it turns out that if you want, you can actually apply the kernel's idea and define the source of features using landmarks and so on for logistic regression.

But the computational tricks that apply for support vector machines don't generalize well to other algorithms like logistic regression. And so, using kernels with logistic regression is going to be very slow, whereas, because of computational tricks, like that embodied and how it modifies this and the details of how the support vector machine software is implemented, support vector machines and kernels tend to go particularly well together. Whereas, logistic regression and kernels, you know, you can do it, but this would run very slowly. And it won't be able to take advantage of advanced optimization techniques that people have figured out for the particular case of running a support vector machine with a kernel.

But all this pertains only to how you actually implement software to minimize the cost function. I will say more about that in the next video, but you really don't need to know about how to write software to minimize this cost function because you can find very good off the shelf software for doing so. And just as, you know, I wouldn't recommend writing code to invert a matrix or to compute a square root, I actually do not recommend writing software to minimize this cost function yourself, but instead to use off the shelf software packages that people have developed and so those software packages already embody these numerical optimization tricks, so you don't really have to worry about them. But one other thing that is worth knowing about is when you're applying a support vector machine, how do you choose the parameters of the support vector machine?

And the last thing I want to do in this video is say a little word about the bias and variance trade offs when using a support vector machine. When using an SVM, one of the things you need to choose is the parameter C which was in the optimization objective, and you recall that C played a role similar to 1 over lambda, where lambda was the regularization parameter we had for logistic regression.

So, if you have a large value of C, this corresponds to what we have back in logistic regression, of a small value of lambda meaning of not using much regularization. And if you do that, you tend to have a hypothesis with lower bias and higher variance. Whereas if you use a smaller value of C then this corresponds to when we are using logistic regression with a large value of lambda and that corresponds to a hypothesis with higher bias and lower variance. And so, a hypothesis with large C has a higher variance, and is more prone to overfitting, whereas a hypothesis with small C has higher bias and is thus more prone to underfitting. So this parameter C is one of the parameters we need to choose.

The other one is the parameter sigma squared, which appeared in the Gaussian kernel. So if the Gaussian kernel sigma squared is large, then in the similarity function, which was this you know  $E$  to the minus  $x$  minus landmark varies squared over  $2\sigma^2$ . In this one of the example; If I have only one feature,  $x_1$ , if I have a landmark there at that location, if  $\sigma^2$  is large, then, you know, the Gaussian kernel would tend to fall off relatively slowly and so this would be my feature  $f(i)$ , and so this would be smoother function that varies more smoothly, and so this will give you a hypothesis with higher bias and lower variance, because the Gaussian kernel that falls off smoothly, you tend to get a hypothesis that varies slowly, or varies smoothly as you change the input  $x$ . Whereas in contrast, if  $\sigma^2$  was small and if that's my landmark given my 1 feature  $x_1$ , you know, my Gaussian kernel, my similarity function, will vary more abruptly.

And in both cases I'd pick out 1, and so if sigma squared is small, then my features vary less smoothly. So if it's just higher slopes or higher derivatives here. And using this, you end up fitting hypotheses of lower bias and you can have higher variance. And if you look at this week's points exercise, you actually get to play around with some of these ideas yourself and see these effects yourself.

So, that was the support vector machine with kernels algorithm. And hopefully this discussion of bias and variance will give you some sense of how you can expect this algorithm to behave as well.

### Using An SVM

So far we've been talking about SVMs in a fairly abstract level.

In this video I'd like to talk about what you actually need to do in order to run or to use an SVM. The support vector machine algorithm poses a particular optimization problem. But as I briefly mentioned in an earlier video, I really do not recommend writing your own software to solve for the parameter's theta yourself. So just as today, very few of us, or maybe almost essentially none of us would think of writing code ourselves to invert a matrix or take a square root of a number, and so on. We just, you know, call some library function to do that. In the same way, the software for solving the SVM optimization problem is very complex, and there have been researchers that have been doing essentially numerical optimization research for many years. So you come up with good software libraries and good software packages to do this. And then strongly recommend just using one of the highly optimized software libraries rather than trying to implement something yourself. And there are lots of good software libraries out there.

The two that I happen to use the most often are the linear SVM but there are really lots of good software libraries for doing this that you know, you can link to many of the major programming languages that you may be using to code up learning algorithm. Even though you shouldn't be writing your own SVM optimization software, there are a few things you need to do, though.

First is to come up with some choice of the parameter's C. We talked a little bit of the bias/variance properties of this in the earlier video. Second, you also need to choose the kernel or the similarity function that you want to use. So one choice might be if we decide not to use any kernel. And the idea of no kernel is also called a linear kernel. So if someone says, I use an SVM with a linear kernel, what that means is you know, they use an SVM without using a kernel and it was a version of the SVM that just uses theta transpose X, right, that predicts 1 theta 0 plus theta 1 X1 plus so on plus theta N, X N is greater than equals 0.

This term linear kernel, you can think of this as you know this is the version of the SVM that just gives you a standard linear classifier. So that would be one reasonable choice for some problems, and you know, there would be many software libraries, like linear, was one example, out of many, one example of a software library that can train an SVM without using a kernel, also called a linear kernel.

So, why would you want to do this? If you have a large number of features, if  $N$  is large, and  $M$  the number of training examples is small, then you know you have a huge number of features that if  $X$ , this is an  $R^n$ ,  $R^{n+1}$ .

So if you have a huge number of features already, with a small training set, you know, maybe you want to just fit a linear decision boundary and not try to fit a very complicated nonlinear function, because might not have enough data. And you might risk overfitting, if you're trying to fit a very complicated function in a very high dimensional feature space, but if your training set sample is small. So this would be one reasonable setting where you might decide to just not use a kernel, or equivalents to use what's called a linear kernel.

A second choice for the kernel that you might make, is this Gaussian kernel, and this is what we had previously. And if you do this, then the other choice you need to make is to choose this parameter sigma squared when we also talk a little bit about the bias variance tradeoffs of how, if sigma squared is large, then you tend to have a higher bias, lower variance classifier, but if sigma squared is small, then you have a higher variance, lower bias classifier. So when would you choose a Gaussian kernel? Well, if your omission of features  $X$ , I mean  $R^n$ , and if  $N$  is small, and, ideally, you know, if  $n$  is large, right, so that's if, you know, we have say, a two-dimensional training set, like the example I drew earlier. So  $n$  is equal to 2, but we have a pretty large training set. So, you know, I've drawn in a fairly large number of training examples, then maybe you want to use a kernel to fit a more complex nonlinear decision boundary, and the Gaussian kernel would be a fine way to do this. I'll say more towards the end of the video, a little bit more about when you might choose a linear kernel, a Gaussian kernel and so on. But if concretely, if you decide to use a Gaussian kernel, then here's what you need to do. Depending on what support vector machine software package you use, it may ask you to implement a kernel function, or to implement the similarity function. So if you're using an octave or MATLAB implementation of an SVM, it may ask you to provide a function to compute a particular feature of the kernel. So this is really computing  $f_{\text{subscript } i}$  for one particular value of  $i$ , where  $f$  here is just a single real number, so maybe I should move this better written  $f(i)$ , but what you need to do is to write a kernel function that takes this input, you know, a training example or a test example whatever it takes in some vector  $X$  and takes as input one of the landmarks and but only I've come down  $X_1$  and  $X_2$  here, because the landmarks are really training examples as well. But what you need to do is write software that takes this input, you know,  $X_1, X_2$  and computes this sort of similarity function between them and return a real number. And so what some support vector machine packages do is expect you to provide this kernel function that take this input you know,  $X_1, X_2$  and returns a real number. And then it will take it from there and it will automatically generate all the features, and so automatically take  $X$  and map it to  $f_1, f_2, \dots, f_m$  using this function that you write, and generate all the features and train the support vector machine from there. But sometimes you do need to provide this function yourself. Other if you are using the Gaussian kernel, some SVM implementations will also include the Gaussian kernel and a few other kernels as well, since the Gaussian kernel is probably the most common kernel. Gaussian and linear kernels are really the two most popular kernels by far.

Just one implementational note. If you have features of very different scales, it is important to perform feature scaling before using the Gaussian kernel. And here's why. If you imagine the computing the norm between  $X$  and  $l$ , right, so this term here, and the numerator term over there.

What this is doing, the norm between  $X$  and  $l$ , that's really saying, you know, let's compute the vector  $V$ , which is equal to  $X$  minus  $l$ . And then let's compute the norm of vector  $V$ , which is the difference between  $X$ . So the norm of  $V$  is really equal to  $V_1^2 + V_2^2 + \dots + V_n^2$ . Because here  $X$  is in  $R^n$ , or  $R^n + 1$ , but I'm going to ignore, you know,  $X_0$ . So, let's pretend  $X$  is an  $R^n$ , square on the left side is what makes this correct. So this is equal to that, right? And so written differently, this is going to be  $X_1^2 - l_1^2 + X_2^2 - l_2^2 + \dots + X_n^2 - l_n^2$ . And now if your features take on very different ranges of value.

So take a housing prediction, for example, if your data is some data about houses. And if  $X$  is in the range of thousands of square feet, for the first feature,  $X_1$ . But if your second feature,  $X_2$  is the number of bedrooms. So if this is in the range of one to five bedrooms, then  $X_1 - l_1$  is going to be huge. This could be like a thousand squared, whereas  $X_2 - l_2$  is going to be much smaller and if that's the case, then in this term, those distances will be almost essentially dominated by the sizes of the houses and the number of bathrooms would be largely ignored. As so as, to avoid this in order to make a machine work well, do perform future scaling. And that will sure that the SVM gives, you know, comparable amount of attention to all of your different features, and not just to in this example to size of houses were big movement here the features.

When you try a support vector machines chances are by far the two most common kernels you use will be the linear kernel, meaning no kernel, or the Gaussian kernel that we talked about. And just one note of warning which is that not all similarity functions you might come up with are valid kernels. And the Gaussian kernel and the linear kernel and other kernels that you sometimes others will use, all of them need to satisfy a technical condition. It's called Mercer's Theorem and the reason you need to this is because support vector machine algorithms or implementations of the SVM have lots of clever numerical optimization tricks.

In order to solve for the parameter's theta efficiently and in the original design envisaged, those are decision made to restrict our attention only to kernels that satisfy this technical condition called Mercer's Theorem. And what that does is, that makes sure that all of these SVM packages, all of these SVM software packages can use the large class of optimizations and get the parameter theta very quickly. So, what most people end up doing is using either the linear or Gaussian kernel, but there are a few other kernels that also satisfy Mercer's theorem and that you may run across other people using, although I personally end up using other kernels you know, very, very rarely, if at all.

Just to mention some of the other kernels that you may run across. One is the polynomial kernel. And for that the similarity between  $X$  and  $l$  is defined as, there are a lot of options, you can take  $X^T l^2$ . So, here's one measure of how similar  $X$  and  $l$  are. If  $X$  and  $l$  are very close with each other, then the inner product will tend to be large. And so, you know, this is a slightly unusual kernel. That is not used that often, but you may run across some people using it. This is one version of a polynomial kernel. Another is  $X^T l^3$ . These are all examples of the polynomial kernel.  $X^T l^2 + 1^3$ .  $X^T l^2 + 5$  maybe a number different than one 5 and, you know, to the power of 4 and so the polynomial kernel actually has two parameters.

One is, what number do you add over here? It could be 0. This is really plus 0 over there, as well as what's the degree of the polynomial over there. So the degree power and these numbers. And the more general form of the polynomial kernel is  $X$  transpose  $l$ , plus some constant and then to some degree in the  $X_1$  and so both of these are parameters for the polynomial kernel. So the polynomial kernel almost always or usually performs worse. And the Gaussian kernel does not use that much, but this is just something that you may run across.

Usually it is used only for data where  $X$  and  $l$  are all strictly non negative, and so that ensures that these inner products are never negative. And this captures the intuition that  $X$  and  $l$  are very similar to each other, then maybe the inter product between them will be large. They have some other properties as well but people tend not to use it much. And then, depending on what you're doing, there are other, sort of more esoteric kernels as well, that you may come across. You know, there's a string kernel, this is sometimes used if your input data is text strings or other types of strings.

There are things like the chi-square kernel, the histogram intersection kernel, and so on. There are sort of more esoteric kernels that you can use to measure similarity between different objects. So for example, if you're trying to do some sort of text classification problem, where the input  $x$  is a string then maybe we want to find the similarity between two strings using the string kernel, but I personally you know end up very rarely, if at all, using these more esoteric kernels. I think I might have used the chi-square kernel, maybe once in my life and the histogram kernel, maybe once or twice in my life. I've actually never used the string kernel myself. But in case you've run across this in other applications. You know, if you do a quick web search we do a quick Google search or quick Bing search you should have found definitions that these are the kernels as well. So just two last details I want to talk about in this video. One in multiclass classification. So, you have four classes or more generally 3 classes output some appropriate decision boundary between your multiple classes. Most SVM, many SVM packages already have built-in multiclass classification functionality. So if you're using a pattern like that, you just use the both that functionality and that should work fine. Otherwise, one way to do this is to use the one versus all method that we talked about when we are developing logistic regression. So what you do is you trade kSVM's if you have  $k$  classes, one to distinguish each of the classes from the rest.

And this would give you  $k$  parameter vectors, so this will give you, up to  $\theta_1$ , which is trying to distinguish class  $y = 1$  from all of the other classes, then you get the second parameter, vector  $\theta_2$ , which is what you get when you, you know, have  $y = 2$  as the positive class and all the others as negative class and so on up to a parameter vector  $\theta_k$ , which is the parameter vector for distinguishing the final class  $y$  from anything else, and then lastly, this is exactly the same as the one versus all method we have for logistic regression. Where we just predict the class  $i$  with the largest  $\theta_i^T X$ . So let's multiclass classification designate.

For the more common cases that there is a good chance that whatever software package you use, you know, there will be a reasonable chance that are already have built in multiclass classification functionality, and so you don't need to worry about this result. Finally, we developed support vector machines starting off with logistic regression and then modifying the cost function a little bit. The last thing we want to do in this video is, just say a little bit about.

when you will use one of these two algorithms, so let's say  $n$  is the number of features and  $m$  is the number of training examples. So, when should we use one algorithm versus the other?

Well, if  $n$  is larger relative to your training set size, so for example, if you take a business with a number of features this is much larger than  $m$  and this might be, for example, if you have a text classification problem, where you know, the dimension of the feature vector is I don't know, maybe, 10 thousand. And if your training set size is maybe 10 you know, maybe, up to 1000. So, imagine a spam classification problem, where email spam, where you have 10,000 features corresponding to 10,000 words but you have, you know, maybe 10 training examples or maybe up to 1,000 examples. So if  $n$  is large relative to  $m$ , then what I would usually do is use logistic regression or use it as the  $m$  without a kernel or use it with a linear kernel.

Because, if you have so many features with smaller training sets, you know, a linear function will probably do fine, and you don't have really enough data to fit a very complicated nonlinear function. Now if  $n$  is small and  $m$  is intermediate what I mean by this is  $n$  is maybe anywhere from 1 - 1000, 1 would be very small. But maybe up to 1000 features and if the number of training examples is maybe anywhere from 10, you know, 10 to maybe up to 10,000 examples. Maybe up to 50,000 examples. If  $m$  is pretty big like maybe 10,000 but not a million. Right? So if  $m$  is an intermediate size then often an SVM with a linear kernel will work well.

We talked about this early as well, with the one concrete example, this would be if you have a two dimensional training set. So, if  $n$  is equal to 2 where you have, you know, drawing in a pretty large number of training examples. So Gaussian kernel will do a pretty good job separating positive and negative classes. One third setting that's of interest is if  $n$  is small but  $m$  is large. So if  $n$  is you know, again maybe 1 to 1000, could be larger. But if  $m$  was, maybe 50,000 and greater to millions. So, 50,000, a 100,000, million, trillion. You have very very large training set sizes, right. So if this is the case, then a SVM of the Gaussian Kernel will be somewhat slow to run. Today's SVM packages, if you're using a Gaussian Kernel, tend to struggle a bit. If you have, you know, maybe 50 thousands okay, but if you have a million training examples, maybe or even a 100,000 with a massive value of  $m$ . Today's SVM packages are very good, but they can still struggle a little bit when you have a massive, massive trainings that size when using a Gaussian Kernel.

So in that case, what I would usually do is try to just manually create have more features and then use logistic regression or an SVM without the Kernel. And in case you look at this slide and you see logistic regression or SVM without a kernel. In both of these places, I kind of paired them together. There's a reason for that, is that logistic regression and SVM without the kernel, those are really pretty similar algorithms and, you know, either logistic regression or SVM without a kernel will usually do pretty similar things and give pretty similar performance, but depending on your implementational details, one may be more efficient than the other.

But, where one of these algorithms applies, logistic regression where SVM without a kernel, the other one is likely to work pretty well as well. But along with the power of the SVM is when you use different kernels to learn complex nonlinear functions. And this regime, you know, when you have maybe up to 10,000 examples, maybe up to 50,000. And your number of features, this is reasonably large. That's a very common regime and maybe that's a regime where a support vector machine with a kernel kernel will shine. You can do things that are much harder to do that

will need logistic regression. And finally, where do neural networks fit in? Well for all of these problems, for all of these different regimes, a well designed neural network is likely to work well as well. The one disadvantage, or the one reason that might not sometimes use the neural network is that, for some of these problems, the neural network might be slow to train. But if you have a very good SVM implementation package, that could run faster, quite a bit faster than your neural network. And, although we didn't show this earlier, it turns out that the optimization problem that the SVM has is a convex optimization problem and so the good SVM optimization software packages will always find the global minimum or something close to it. And so for the SVM you don't need to worry about local optima. In practice local optima aren't a huge problem for neural networks but they all solve, so this is one less thing to worry about if you're using an SVM. And depending on your problem, the neural network may be slower, especially in this sort of regime than the SVM. In case the guidelines they gave here, seem a little bit vague and if you're looking at some problems, you know, the guidelines are a bit vague, I'm still not entirely sure, should I use this algorithm or that algorithm, that's actually okay.

When I face a machine learning problem, you know, sometimes its actually just not clear whether that's the best algorithm to use, but as you saw in the earlier videos, really, you know, the algorithm does matter, but what often matters even more is things like, how much data do you have. And how skilled are you, how good are you at doing error analysis and debugging learning algorithms, figuring out how to design new features and figuring out what other features to give you learning algorithms and so on. And often those things will matter more than what you are using logistic regression or an SVM. But having said that, the SVM is still widely perceived as one of the most powerful learning algorithms, and there is this regime of when there's a very effective way to learn complex non linear functions.

And so I actually, together with logistic regressions, neural networks, SVM's, using those to speed learning algorithms you're I think very well positioned to build state of the art you know, machine learning systems for a wide region for applications and this is another very powerful tool to have in your arsenal. One that is used all over the place in Silicon Valley, or in industry and in the Academia, to build many high performance machine learning system.