

Logistic Regression Model

Cost Function (Video)

In this video, we'll talk about how to fit the parameters of θ for the 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 logistic regression model. We have a training set of m training examples and as usual, each of our examples is represented by a vector that's $n + 1$ dimensional, and as usual we have $x_0 = 1$. First feature or a zero feature is always equal to one. And because this is a classification problem, our training set has the property that every label y is either 0 or 1. This is a hypothesis, and the parameters of a hypothesis is this θ 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 θ ?

Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$

m examples

$$x \in \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_n \end{bmatrix} \quad \mathbb{R}^{n+1} \quad \underline{x_0 = 1, y \in \{0, 1\}}$$

$$h_{\theta}(x) = \frac{1}{1 + e^{-\theta^T x}}$$

How to choose parameters θ ?

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 h of x comma y to be equal to one half of this squared error. 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 h of x , and the actual label was 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 have, 0 for 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 data. Here's what I mean by non-convex. Have some cross function j of θ and for logistic regression, this function h here has a nonlinearity that is one over one plus e to the negative θ transpose. So this is a pretty complicated nonlinear function. And if you take the function, plug it in here. And then take this cost function and plug it in there and then plot what j of θ looks like. You find that j of θ 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 θ that is convex, that is a single bow-shaped function that looks like this so that if you run θ in the we would be guaranteed that would converge to the global minimum. And the problem with using this great cost function is that because of this very nonlinear function that appears in the middle here, J of θ ends up being a nonconvex function if you were to define it as a square cost function. So what

we'd like to do is, 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. Back when we were developing the linear regression model, we used the following cost function. I've written this slightly differently where instead of $\frac{1}{2m}$, I've taken a one-half and put it inside the summation instead. 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 costs of h of x , y and I'm going to define that total cost of h of x , y 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 $\frac{1}{n}$ 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 h of x comma y to be equal to one half of this squared error. 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 h of x , and the actual label was 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 have, 0 for 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 data. Here's what I mean by non-convex. Have some cross function j of θ and for logistic regression, this function h here has a nonlinearity that is one over one plus e to the negative θ transpose. So this is a pretty complicated nonlinear function. And if you take the function, plug it in here. And then take this cost function and plug it in there and then plot what j of θ looks like. You find that j of θ 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 θ that is convex, that is a single bow-shaped function that looks like this so that if you run θ in the we would be guaranteed that would converge to the global minimum. And the problem with using this great cost function is that because of this very nonlinear function that appears in the middle here, J of θ ends up being a nonconvex function if you were to define it as a square cost function. So what we'd like to do is, 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.

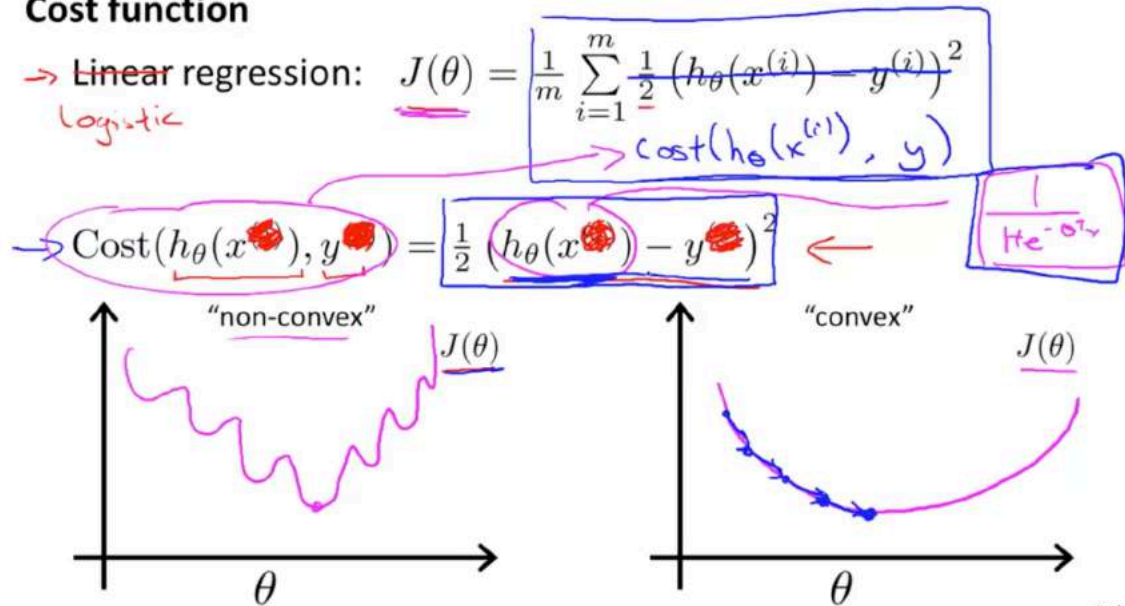
Cost function

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

$\text{cost}(h_{\theta}(x^{(i)}), y)$

$$\text{Cost}(h_{\theta}(x^{(i)}), y^{(i)}) = \frac{1}{2} (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

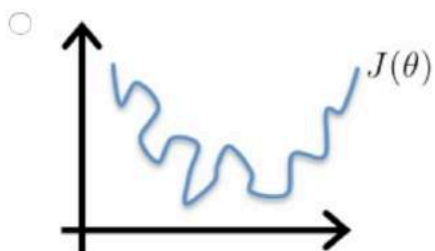
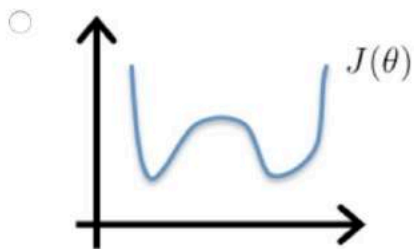
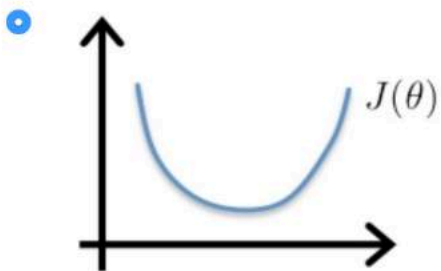
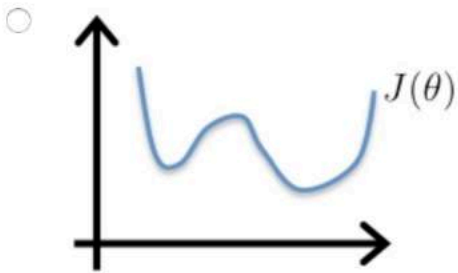
Cost function



Andrew I

Question

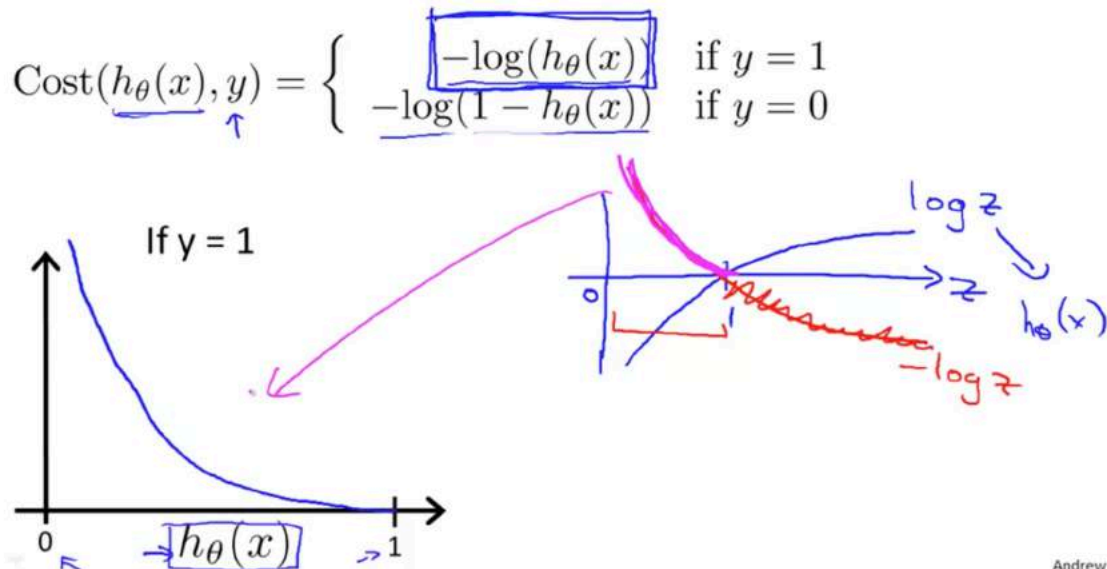
Consider minimizing a cost function $J(\theta)$. Which one of these functions is convex?



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 of 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 $y = 0$. This looks like a pretty complicated function, but let's plot this function to gain some intuition about what it's doing. 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 of 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.

Logistic regression cost function



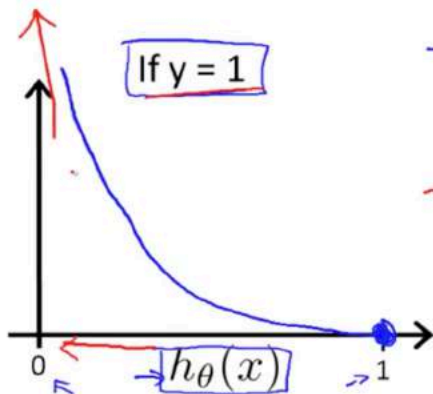
Andrew B

Now, this cost function has a few interesting and desirable properties. 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 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

of 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 Let's look at what the cost function looks like for y equals 0. 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 consider the case of y equals 1.

Logistic regression cost function

$$\text{Cost}(h_{\theta}(x), y) = \begin{cases} -\log(h_{\theta}(x)) & \text{if } y = 1 \\ -\log(1 - h_{\theta}(x)) & \text{if } y = 0 \end{cases}$$



→ Cost = 0 if $y = 1, h_{\theta}(x) = 1$

But as $h_{\theta}(x) \rightarrow 0$
 $\text{Cost} \rightarrow \infty$

→ Captures intuition that if $h_{\theta}(x) = 0$, (predict $P(y = 1|x; \theta) = 0$), but $y = 1$, we'll penalize learning algorithm by a very large cost.

Question

In logistic regression, the cost function for our hypothesis outputting (predicting) $h_{\theta}(x)$ on a training example that has label $y \in \{0, 1\}$ is:

$$\text{cost}(h_{\theta}(x), y) = \begin{cases} -\log h_{\theta}(x) & \text{if } y = 1 \\ -\log(1 - h_{\theta}(x)) & \text{if } y = 0 \end{cases}$$

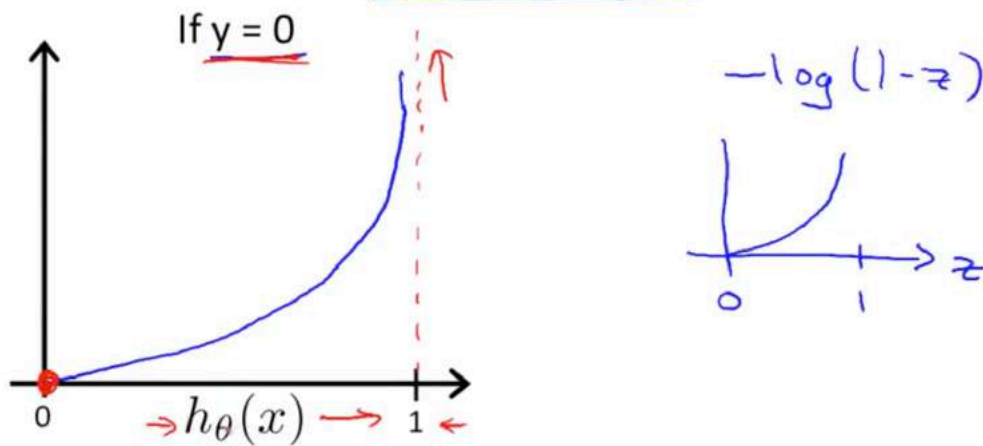
Which of the following are true? Check all that apply.

- ☒ If $h_{\theta}(x) = y$, then $\text{cost}(h_{\theta}(x), y) = 0$ (for $y = 0$ and $y = 1$).
- ☒ If $y = 0$, then $\text{cost}(h_{\theta}(x), y) \rightarrow \infty$ as $h_{\theta}(x) \rightarrow 1$.
- ☐ If $y = 0$, then $\text{cost}(h_{\theta}(x), y) \rightarrow \infty$ as $h_{\theta}(x) \rightarrow 0$.
- ☒ Regardless of whether $y = 0$ or $y = 1$, if $h_{\theta}(x) = 0.5$, then $\text{cost}(h_{\theta}(x), y) > 0$.

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, probably 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 melted. The predicted y of z 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.

Logistic regression cost function

$$\text{Cost}(h_{\theta}(x), y) = \begin{cases} -\log(h_{\theta}(x)) & \text{if } y = 1 \\ -\log(1 - h_{\theta}(x)) & \text{if } y = 0 \end{cases}$$



In this video, we will define the cost function for a single train example. The topic of convexity analysis is now beyond the scope of this course, but it is possible to show that with a particular choice of cost function, this will give a convex optimization problem. Overall cost function J of θ 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.

Cost Function (Transcript)

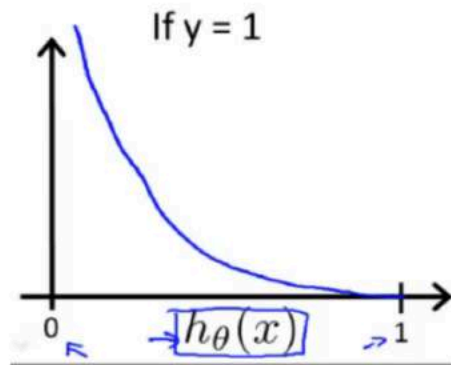
We cannot use the same cost function that we use for linear regression because the Logistic Function will cause the output to be wavy, causing many local optima. In other words, it will not be a convex function.

Instead, our cost function for logistic regression looks like:

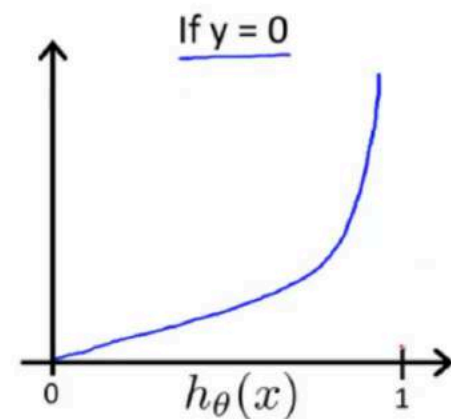
$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \text{Cost}(h_{\theta}(x^{(i)}), y^{(i)})$$

$$\begin{aligned} \text{Cost}(h_{\theta}(x), y) &= -\log(h_{\theta}(x)) && \text{if } y = 1 \\ \text{Cost}(h_{\theta}(x), y) &= -\log(1 - h_{\theta}(x)) && \text{if } y = 0 \end{aligned}$$

When $y = 1$, we get the following plot for $J(\theta)$ vs $h_{\theta}(x)$:



Similarly, when $y = 0$, we get the following plot for $J(\theta)$ vs $h_{\theta}(x)$:



$$\begin{aligned} \text{Cost}(h_{\theta}(x), y) &= 0 \text{ if } h_{\theta}(x) = y \\ \text{Cost}(h_{\theta}(x), y) &\rightarrow \infty \text{ if } y = 0 \text{ and } h_{\theta}(x) \rightarrow 1 \\ \text{Cost}(h_{\theta}(x), y) &\rightarrow \infty \text{ if } y = 1 \text{ and } h_{\theta}(x) \rightarrow 0 \end{aligned}$$

If our correct answer 'y' is 0, then the cost function will be 0 if our hypothesis function also outputs 0. If our hypothesis approaches 1, then the cost function will approach infinity.

If our correct answer 'y' is 1, then the cost function will be 0 if our hypothesis function outputs 1. If our hypothesis approaches 0, then the cost function will approach infinity.

Note that writing the cost function in this way guarantees that $J(\theta)$ is convex for logistic regression.

Simplified Cost function and Gradient descent (Video)

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. 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 zero or one, right? That's sort of part of the mathematical definition of y . Because y is either zero or one, 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. We say that cost of $H(x), y$. 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, no, 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. We know that there are only two possible cases. y must be zero or one. 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 $1 - y$ is going to be equal to 0, right. So if y is equal to 1, then $1 - y$ is $1 - 1$, 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 $y \times \log h(x)$. 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 $1 - y$, if y is equal to zero would be equal to 1, because $1 - y$ becomes $1 - 0$ 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.

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 can therefore write all our cost functions for logistic regression as follows. 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 of theta. So if we try to minimize this, this would give us some set of parameters 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 as estimating the probability that y is equal to one. So all that remains to be done is figure out how to actually minimize J of theta as a function of theta so that we can actually fit the parameters to our training set.

Logistic regression cost function

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \text{Cost}(h_{\theta}(x^{(i)}), y^{(i)})$$
$$= -\frac{1}{m} \left[\sum_{i=1}^m y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_{\theta}(x^{(i)})) \right]$$

To fit parameters θ :

$$\min_{\theta} J(\theta) \quad \text{Get } \underline{\theta}$$

To make a prediction given new x :

$$\text{Output } \underline{h_{\theta}(x)} = \frac{1}{1 + e^{-\theta^T x}}$$

$$\underline{p(y=1 | x; \theta)}$$

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 xij. 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

$$\rightarrow J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^m y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_{\theta}(x^{(i)})) \right]$$

Want $\min_{\theta} J(\theta)$:

$$\left[\begin{array}{l} \text{Repeat } \{ \\ \quad \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \\ \} \end{array} \right]$$

(simultaneously update all θ_j)

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

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 θ , which with parameters $\theta_0, \theta_1, \theta_2$, down to θ_n . And you will use this update to simultaneously update all of your values of θ . 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 $\theta^T X$, 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, θ_0 down to θ_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 $n + 1$. So update each of these parameter values in turn. But of course rather than using a for loop, ideally we would also use a vector rise implementation. So that a vector rise implementation can update all of these $n + 1$ parameters all in one fell swoop. And to check your own understanding, you might see if you can figure out how to do the vector rise implementation with this algorithm as well.

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 θ_j)

}

$$\Theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} \leftarrow \text{for } i = 0 \text{ to } n$$

$$h_{\theta}(x) = \Theta^T x$$

$$\rightarrow h_{\theta}(x) = \frac{1}{1 + e^{-\Theta^T x}}$$

Algorithm looks identical to linear regression!

Question

Suppose you are running gradient descent to fit a logistic regression model with parameter $\theta \in \mathbb{R}^{n+1}$. Which of the following is a reasonable way to make sure the learning rate α is set properly and that gradient descent is running correctly?

- ☐ Plot $J(\theta) = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$ as a function of the number of iterations (i.e. the horizontal axis is the iteration number) and make sure $J(\theta)$ is decreasing on every iteration.
- ☒ Plot $J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))]$ as a function of the number of iterations and make sure $J(\theta)$ is decreasing on every iteration.
- ☐ Plot $J(\theta)$ as a function of θ and make sure it is decreasing on every iteration.
- ☐ Plot $J(\theta)$ as a function of θ and make sure it is convex.

One iteration of gradient descent simultaneously performs these updates:

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

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

\vdots

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

We would like a vectorized implementation of the form $\theta := \theta - \alpha \delta$ (for some vector $\delta \in \mathbb{R}^{n+1}$).

What should the vectorized implementation be?

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

Correct

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

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

☐ All of the above are correct implementations.

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 gradient 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.

Simplified Cost function and Gradient descent (Transcript)

Note: [6:53 - the gradient descent equation should have a $1/m$ factor]

We can compress our cost function's two conditional cases into one case:

$$\text{Cost}(h_{\theta}(x), y) = -y \log(h_{\theta}(x)) - (1 - y) \log(1 - h_{\theta}(x))$$

Notice that when y is equal to 1, then the second term $(1 - y) \log(1 - h_{\theta}(x))$ will be zero and will not affect the result. If y is equal to 0, then the first term $-y \log(h_{\theta}(x))$ will be zero and will not affect the result.

We can fully write out our entire cost function as follows:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))]$$

A vectorized implementation is:

$$h = g(X\theta)$$
$$J(\theta) = \frac{1}{m} \cdot (-y^T \log(h) - (1 - y)^T \log(1 - h))$$

Gradient Descent

Remember that the general form of gradient descent is:

$$\begin{aligned} &\text{Repeat } \{ \\ &\quad \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \\ &\} \end{aligned}$$

We can work out the derivative part using calculus to get:

$$\begin{aligned} &\text{Repeat } \{ \\ &\quad \theta_j := \theta_j - \frac{\alpha}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \\ &\} \end{aligned}$$

Notice that this algorithm is identical to the one we used in linear regression. We still have to simultaneously update all values in θ .

A vectorized implementation is:

$$\theta := \theta - \frac{\alpha}{m} X^T (g(X\theta) - y)$$

Advanced Optimization (Video)

In the last video, we talked about gradient descent for minimizing the cost

function J of θ 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 θ and they can compute two things: J of θ and these partial derivative terms for, you know, J equals 0, 1 up to N . 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 θ . So another way of thinking about gradient descent is that we need to supply code to compute J of θ and these derivatives, and 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 θ . 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.

Optimization algorithm

Cost function $J(\theta)$. Want $\min_{\theta} J(\theta)$.

Given θ , we have code that can compute

$$\begin{aligned} \rightarrow & - J(\theta) \\ \rightarrow & - \frac{\partial}{\partial \theta_j} J(\theta) \quad (\text{for } j = 0, 1, \dots, n) \end{aligned}$$

Gradient descent:

Repeat {

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

}

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 θ , 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 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. These three algorithms have a number of advantages. One is that, with any of this algorithms you usually do not need to manually pick the learning rate α . 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 inter-loop. And, in fact, they have a clever inter-loop called a line search algorithm that automatically tries out different values for the learning rate α and automatically picks a good learning rate α so that it can even pick a different learning rate for every iteration. And so then you don't need to choose it yourself. 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 O-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 inter-loop of what these algorithms do. If these algorithms have a disadvantage, I'd say that 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.

Optimization algorithm

Given θ , we have code that can compute

$$\begin{bmatrix} - J(\theta) \\ - \frac{\partial}{\partial \theta_j} J(\theta) \end{bmatrix} \quad (\text{for } j = 0, 1, \dots, n)$$

Optimization algorithms:

- Gradient descent
- Conjugate gradient
- BFGS
- L-BFGS

Advantages:

- No need to manually pick α
- Often faster than gradient descent.

Disadvantages:

- More complex

So now let's explain how to use these algorithms, I'm going to do so with an example. Let's say that you have a problem with two parameters equals theta zero and theta one. And let's say your cost function is J of 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 of theta as a function of theta. The value that minimizes it is going to be theta 1 equals 5, theta 2 equals 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. So if you want to apply one of the advanced optimization algorithms to minimize cost function J . So, you know, if we didn't know the minimum was at 5, 5, but if you want to have a cost function 5 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. 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 is a options as 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 at 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 for 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.

Example: $\min_{\theta} J(\theta)$

$\rightarrow \theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$ $\theta_1=5, \theta_2=5.$

$\rightarrow J(\theta) = (\theta_1 - 5)^2 + (\theta_2 - 5)^2$

$\rightarrow \frac{\partial}{\partial \theta_1} J(\theta) = 2(\theta_1 - 5)$

$\rightarrow \frac{\partial}{\partial \theta_2} J(\theta) = 2(\theta_2 - 5)$

```
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);
```

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 line. 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. 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

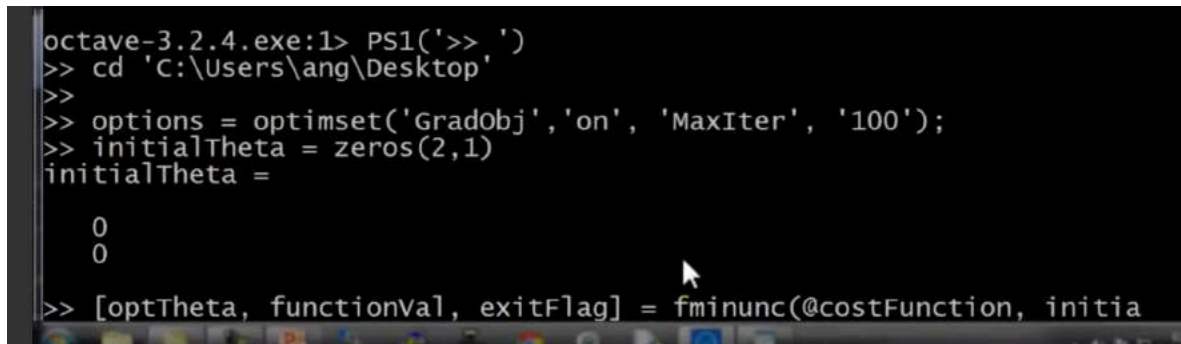
theta. And now I have of theta, function val exit flag equals fminunc constraint. A pointer to the cost function. and provide my initial guess. And the options like so. And if I hit enter this will run the optimization algorithm. And it returns pretty quickly. This funny formatting that's because my 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.

```
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);

|
```



```
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

>> [optTheta, functionVal, exitFlag] = fminunc(@costFunction, initia
```



```

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
>>

```

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.

Example: $\min_{\theta} J(\theta)$

$\rightarrow \theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$ $\theta_1=5, \theta_2=5.$

$\rightarrow J(\theta) = (\theta_1 - 5)^2 + (\theta_2 - 5)^2$

$\rightarrow \frac{\partial}{\partial \theta_1} J(\theta) = 2(\theta_1 - 5)$

$\rightarrow \frac{\partial}{\partial \theta_2} J(\theta) = 2(\theta_2 - 5)$

```

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);

```

$\theta \in \mathbb{R}^d \quad d \geq 2.$

So, that's how we optimize our trial example of this simple quick driving cost function. However, we apply this to let's just say progression. In logistic

regression we have a parameter vector θ , 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 θ comprises these parameters θ_0 through θ_n because octave indexes, vectors using indexing from 1, you know, θ_0 is actually written θ_1 in octave, θ_1 is gonna be written. So, if θ_2 in octave and that's gonna be a written θ_{n+1} , right? And that's because Octave indexes is vectors starting from index of 1 and so the index of 0. So what we need to do then is write a cost function that captures the cost function for logistic regression. Concretely, the cost function needs to return J -val, which is, you know, J -val as you need some codes to compute J of θ and we also need to give it the gradient. So, gradient 1 is going to be some code to compute the partial derivative in respect to θ_0 , the next partial derivative respect to θ_1 and so on. Once again, this is gradient 1, gradient 2 and so on, rather than gradient 0, gradient 1 because octave indexes is 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.

$$\underline{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} \begin{array}{l} \text{theta}(1) \leftarrow \\ \text{theta}(2) \\ \text{theta}(n+1) \end{array}$$

```
function [jVal, gradient] = costFunction(theta)
    jVal = [code to compute  $J(\theta)$ ];
    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  $\frac{\partial}{\partial \theta_n} J(\theta)$ ];
```

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.

Question

Suppose you want to use an advanced optimization algorithm to minimize the cost function for logistic regression with parameters θ_0 and θ_1 . You write the following code:

```
function [jVal, gradient] = costFunction(theta)
    jVal = % code to compute J(theta)
    gradient(1) = CODE#1 % derivative for theta_0
    gradient(2) = CODE#2 % derivative for theta_1
```

What should CODE#1 and CODE#2 above compute?

- ☐ CODE#1 and CODE#2 should compute $J(\theta)$.
- ☐ CODE#1 should be $\theta(1)$ and CODE#2 should be $\theta(2)$.
- ☒ CODE#1 should compute $\frac{1}{m} \sum_{i=1}^m [(h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)}] (= \frac{\partial}{\partial \theta_0} J(\theta))$, and
CODE#2 should compute $\frac{1}{m} \sum_{i=1}^m [(h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x_1^{(i)}] (= \frac{\partial}{\partial \theta_1} J(\theta))$.

Correct

- ☐ None of the above.

Advanced Optimization (Transcript)

Note: [7:35 - '100' should be 100 instead. The value provided should be an integer and not a character string.]

"Conjugate gradient", "BFGS", and "L-BFGS" are more sophisticated, faster ways to optimize θ that can be used instead of gradient descent. We suggest that you should not write these more sophisticated algorithms yourself (unless you are an expert in numerical computing) but use the libraries instead, as they're already tested and highly optimized. Octave provides them.

We first need to provide a function that evaluates the following two functions for a given input value θ :

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

We can write a single function that returns both of these:

```
1 function [jVal, gradient] = costFunction(theta)
2     jVal = [...code to compute J(theta)...];
3     gradient = [...code to compute derivative of J(theta)...];
4 end
```

We can write a single function that returns both of these:

```
1 function [jVal, gradient] = costFunction(theta)
2     jVal = [...code to compute J(theta)...];
3     gradient = [...code to compute derivative of J(theta)...];
4 end
```

Then we can use octave's "fminunc()" optimization algorithm along with the "optimset()" function that creates an object containing the options we want to send to "fminunc()". (Note: the value for MaxIter should be an integer, not a character string - errata in the video at 7:30)

```
1 options = optimset('GradObj', 'on', 'MaxIter', 100);
2 initialTheta = zeros(2,1);
3 [optTheta, functionVal, exitFlag] = fminunc(@costFunction, initialTheta,
4     options);
```

We give to the function "fminunc()" our cost function, our initial vector of theta values, and the "options" object that we created beforehand.

Multiclass Classification

Multiclass Classification: One-vs-all (Video)

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. 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 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.

Multiclass classification

Email foldering/tagging: Work, Friends, Family, Hobby

$y=1$ $y=2$ $y=3$ $y=4$

Medical diagrams: Not ill, Cold, Flu

$y=1$ 2 3

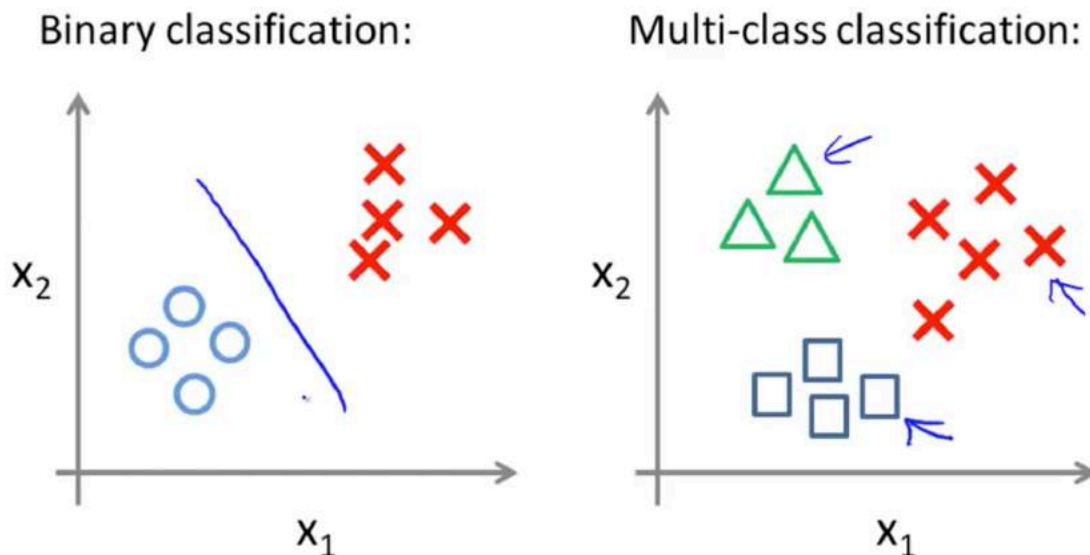
Weather: Sunny, Cloudy, Rain, Snow

$y=1$ 2 3 4 ←

0 1 2 3

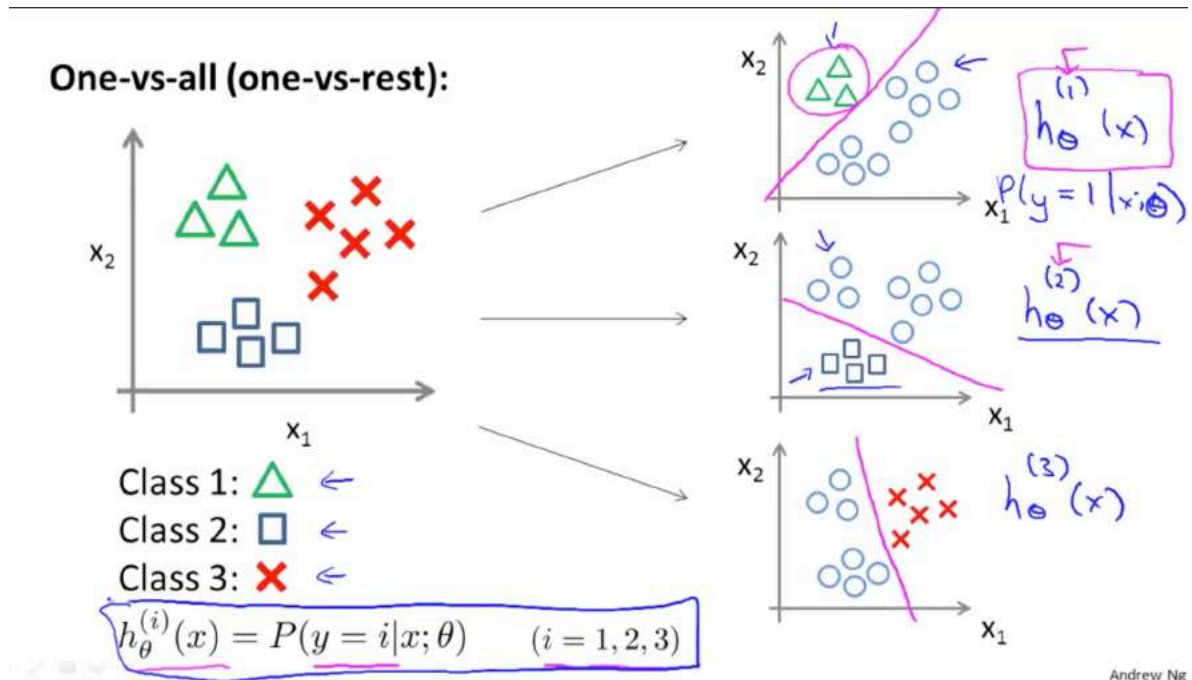
Whereas previously for a binary classification problem, our data sets look like this. For a multi-class classification problem our data sets may look like this where here I'm using three different symbols to represent our three classes. So

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 a regression. We know how to you know maybe fit a straight line to set for 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_{θ^1} of 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_{θ^2} of x , where the superscript two denotes that we're now doing this,

treating the square class as the positive class. And maybe we get classified like that. And finally, we do the same thing for the third class and fit a third classifier $h^{(3)}$ of x , and maybe this will give us a decision boundary of the visible cross fire. 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 x superscript i subscript θ of x . Thus trying to estimate what is the probability that y is equal to class i , given x and parametrized by θ . 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 x superscript one is essentially trying to estimate what is the probability that the y is equal to one, given that x is parametrized by θ . 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.



Question

Suppose you have a multi-class classification problem with k classes (so $y \in \{1, 2, \dots, k\}$). Using the 1-vs.-all method, how many different logistic regression classifiers will you end up training?

- ☐ $k - 1$
- ☒ k
- ☐ $k + 1$
- ☐ Approximately $\log_2(k)$

Just to summarize, what we've done is we want to train a logistic regression classifier $h^{(i)}$ 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.

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

$$\max_i h_{\theta}^{(i)}(x)$$

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

Multiclass Classification: One-vs-all (Transcript)

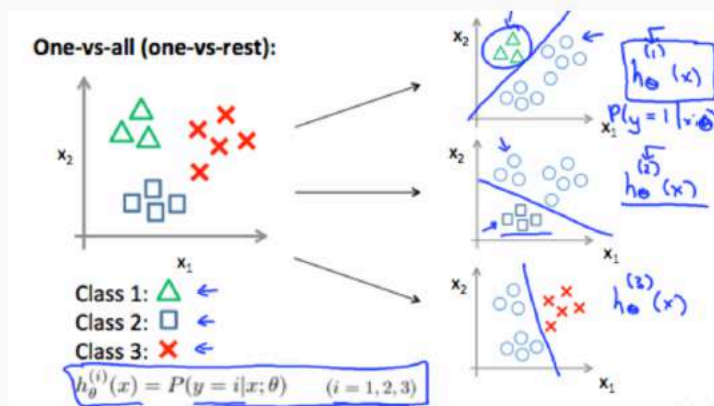
Now we will approach the classification of data when we have more than two categories. Instead of $y = \{0,1\}$ we will expand our definition so that $y = \{0,1 \dots n\}$.

Since $y = \{0,1 \dots n\}$, we divide our problem into $n+1$ (+1 because the index starts at 0) binary classification problems; in each one, we predict the probability that 'y' is a member of one of our classes.

$$\begin{aligned} y &\in \{0, 1, \dots, n\} \\ h_{\theta}^{(0)}(x) &= P(y = 0|x; \theta) \\ h_{\theta}^{(1)}(x) &= P(y = 1|x; \theta) \\ &\dots \\ h_{\theta}^{(n)}(x) &= P(y = n|x; \theta) \\ \text{prediction} &= \max_i (h_{\theta}^{(i)}(x)) \end{aligned}$$

We are basically choosing one class and then lumping all the others into a single second class. We do this repeatedly, applying binary logistic regression to each case, and then use the hypothesis that returned the highest value as our prediction.

The following image shows how one could classify 3 classes:



To summarize:

Train a logistic regression classifier $h_{\theta}(x)$ for each class to predict the probability that $y = i$.

To make a prediction on a new x , pick the class that maximizes $h_{\theta}(x)$

Review

QUIZ

1
point

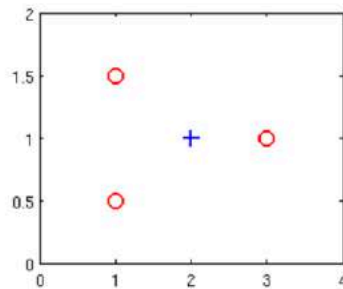
1. Suppose that you have trained a logistic regression classifier, and it outputs on a new example x a prediction $h_{\theta}(x) = 0.7$. This means (check all that apply):

- ☒ Our estimate for $P(y = 1|x; \theta)$ is 0.7.
- ☐ Our estimate for $P(y = 1|x; \theta)$ is 0.3.
- ☐ Our estimate for $P(y = 0|x; \theta)$ is 0.7.
- ☒ Our estimate for $P(y = 0|x; \theta)$ is 0.3.

1
point

2. Suppose you have the following training set, and fit a logistic regression classifier $h_{\theta}(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$.

x_1	x_2	y
1	0.5	0
1	1.5	0
2	1	1
3	1	0



Which of the following are true? Check all that apply.

- ☒ $J(\theta)$ will be a convex function, so gradient descent should converge to the global minimum.
- ☒ Adding polynomial features (e.g., instead using $h_{\theta}(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_1 x_2 + \theta_5 x_2^2)$) could increase how well we can fit the training data.
- ☐ The positive and negative examples cannot be separated using a straight line. So, gradient descent will fail to converge.
- ☐ Because the positive and negative examples cannot be separated using a straight line, linear regression will perform as well as logistic regression on this data.

1
point

3. For logistic regression, the gradient is given by $\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$. Which of these is a correct gradient descent update for logistic regression with a learning rate of α ? Check all that apply.

- ☐ $\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (\theta^T x - y^{(i)}) x_j^{(i)}$ (simultaneously update for all j).
- ☒ $\theta := \theta - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x^{(i)}$.
- ☐ $\theta := \theta - \alpha \frac{1}{m} \sum_{i=1}^m (\theta^T x - y^{(i)}) x^{(i)}$.
- ☒ $\theta := \theta - \alpha \frac{1}{m} \sum_{i=1}^m \left(\frac{1}{1 + e^{-\theta^T x^{(i)}}} - y^{(i)} \right) x^{(i)}$.

1
point

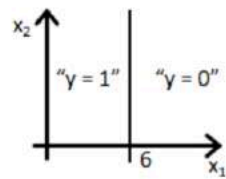
4. Which of the following statements are true? Check all that apply.

- ☐ The cost function $J(\theta)$ for logistic regression trained with $m \geq 1$ examples is always greater than or equal to zero.
- ☐ Since we train one classifier when there are two classes, we train two classifiers when there are three classes (and we do one-vs-all classification).
- ☒ The one-vs-all technique allows you to use logistic regression for problems in which each $y^{(i)}$ comes from a fixed, discrete set of values.
- ☒ For logistic regression, sometimes gradient descent will converge to a local minimum (and fail to find the global minimum). This is the reason we prefer more advanced optimization algorithms such as fminunc (conjugate gradient/BFGS/L-BFGS/etc).

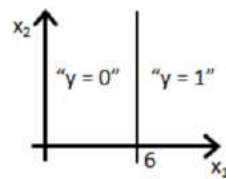
1
point

5. Suppose you train a logistic classifier $h_{\theta}(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$. Suppose $\theta_0 = -6, \theta_1 = 0, \theta_2 = 1$. Which of the following figures represents the decision boundary found by your classifier?

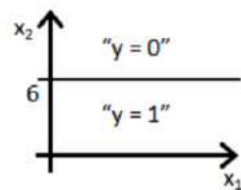
☐ Figure:



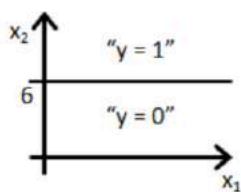
☐ Figure:



☐ Figure:



☒ Figure:



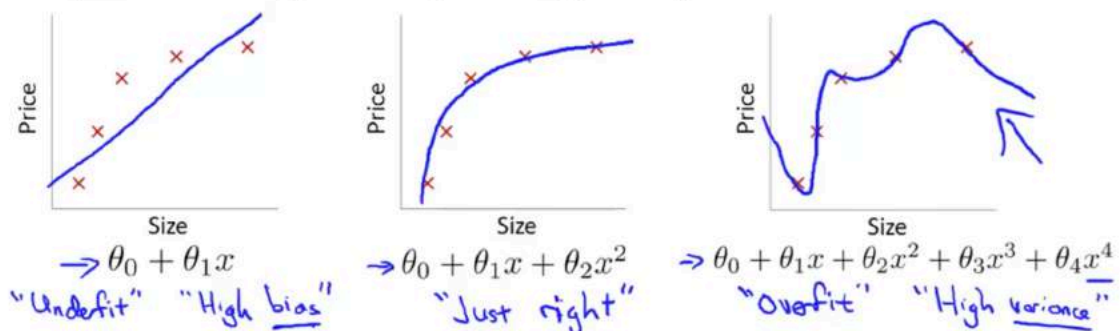
Solving the Problem of Overfitting

The Problem of Overfitting (Video)

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. 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. 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 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 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 of the contrary is preconceptions still are bias, still closes 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 functions enter 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, θ_0 through θ_4 , and, with that, we can actually fill a curve that process 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 is processed 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 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 to learn hypothesis may fit the training side 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 as 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.

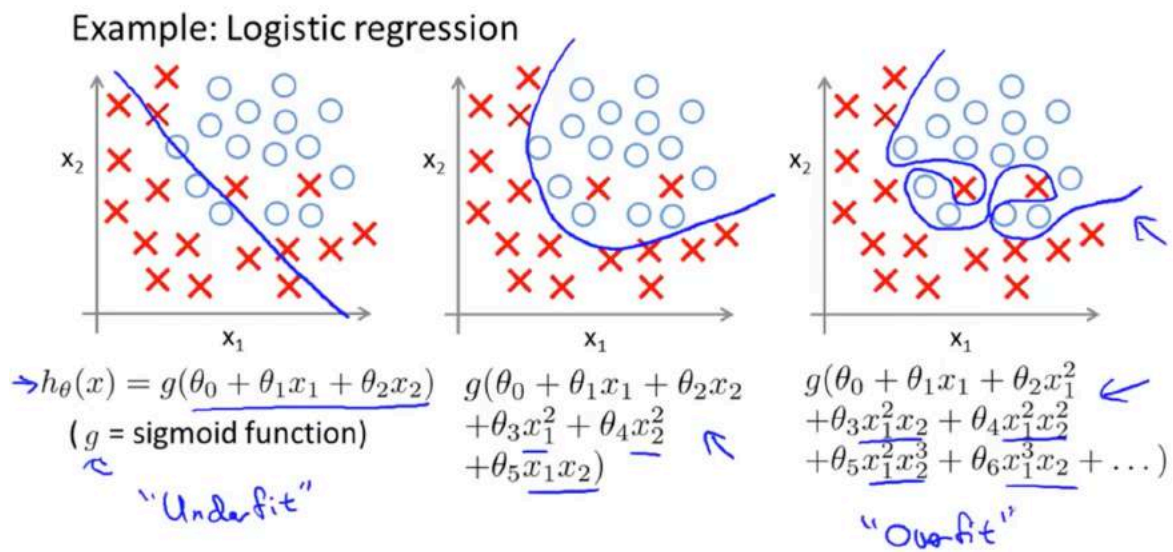
Example: Linear regression (housing prices)



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).

Here is a logistic regression example with two features X_1 and x_2 . 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.



Question

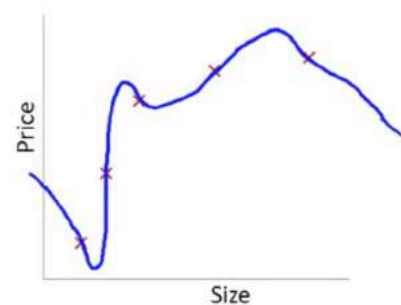
Consider the medical diagnosis problem of classifying tumors as malignant or benign. If a hypothesis $h_{\theta}(x)$ has overfit the training set, it means that:

- ☐ It makes accurate predictions for examples in the training set and generalizes well to make accurate predictions on new, previously unseen examples.
- ☐ It does not make accurate predictions for examples in the training set, but it does generalize well to make accurate predictions on new, previously unseen examples.
- ☒ It makes accurate predictions for examples in the training set, but it does not generalize well to make accurate predictions on new, previously unseen examples.
- ☐ It does not make accurate predictions for examples in the training set and does not generalize well to make accurate predictions on new, previously unseen 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. 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.

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
 \vdots
 x_{100}



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. 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 course, 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. Here, we're going to keep all the features, but we're going to reduce the magnitude or the values of the parameters θ_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.

Addressing overfitting:

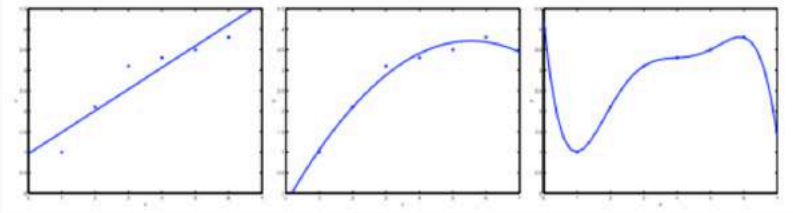
Options:

1. Reduce number of features.
 - — Manually select which features to keep.
 - — Model selection algorithm (later in course).
2. Regularization.
 - — Keep all the features, but reduce magnitude/values of parameters θ_j .
 - Works well when we have a lot of features, each of which contributes a bit to predicting y .

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.

The Problem of Overfitting (Transcript)

Consider the problem of predicting y from $x \in \mathbb{R}$. The leftmost figure below shows the result of fitting a $y = \theta_0 + \theta_1 x$ to a dataset. We see that the data doesn't really lie on straight line, and so the fit is not very good.



Instead, if we had added an extra feature x^2 , and fit $y = \theta_0 + \theta_1 x + \theta_2 x^2$, then we obtain a slightly better fit to the data (See middle figure). Naively, it might seem that the more features we add, the better. However, there is also a danger in adding too many features: The rightmost figure is the result of fitting a 5th order polynomial $y = \sum_{j=0}^5 \theta_j x^j$. We see that even though the fitted curve passes through the data perfectly, we would not expect this to be a very good predictor of, say, housing prices (y) for different living areas (x). Without formally defining what these terms mean, we'll say the figure on the left shows an instance of **underfitting**—in which the data clearly shows structure not captured by the model—and the figure on the right is an example of **overfitting**.

Underfitting, or high bias, is when the form of our hypothesis function h maps poorly to the trend of the data. It is usually caused by a function that is too simple or uses too few features. At the other extreme, overfitting, or high variance, is caused by a hypothesis function that fits the available data but does not generalize well to predict new data. It is usually caused by a complicated function that creates a lot of unnecessary curves and angles unrelated to the data.

This terminology is applied to both linear and logistic regression. There are two main options to address the issue of overfitting:

1) Reduce the number of features:

- Manually select which features to keep.
- Use a model selection algorithm (studied later in the course).

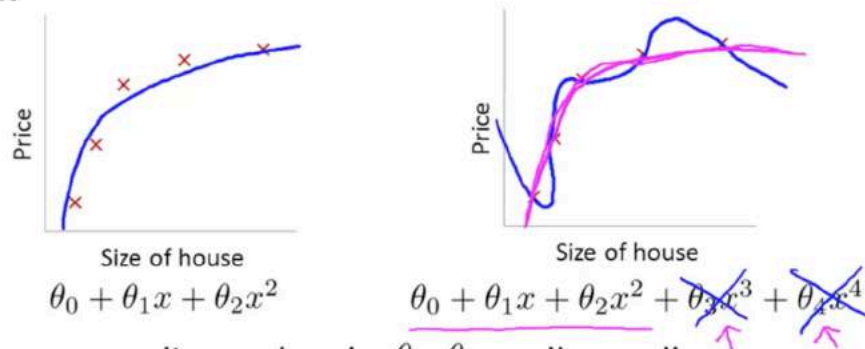
2) Regularization

- Keep all the features, but reduce the magnitude of parameters θ_j .
- Regularization works well when we have a lot of slightly useful features.

Cost Function (Video)

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 self 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. Consider the following, suppose we were to penalize, and, make the parameters θ_3 and θ_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 θ_3 squared, plus 1000 θ_4 squared. 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 θ_3 and θ_4 are small, right? Because otherwise, if you have a thousand times θ_3 , this new cost function's gonna be big. So when we minimize this new function we are going to end up with θ_3 close to 0 and θ_4 close to 0, and as if we're getting rid of these two terms over there. And if we do that, well then, if θ_3 and θ_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, θ_3 , θ_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.

Intuition



Suppose we penalize and make θ_3, θ_4 really small.

$$\rightarrow \min_{\theta} \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + 1000 \theta_3^2 + 1000 \theta_4^2$$

$\theta_3 \approx 0$ $\theta_4 \approx 0$

More generally, here is the idea behind 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 θ_3 and θ_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 as close as you 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 yourself and see it for yourself. But I hope that the example of having θ_3 and θ_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. Lets look at the specific example. For housing price prediction we may have our hundred features that we talked about where 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 θ_3 , θ_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 try to pick, to try to shrink. So, in regularization, what we're going to do, is take our cost function, here's 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 θ_1 , θ_2 , θ_3 up to θ_{100} . By the way, by convention the summation here starts from one so I am not actually going penalize θ_0 being large. That sort of the convention that, the sum $\sum_{i=1}^N$, rather than $\sum_{i=0}^N$. But in practice, it makes very little difference, and, whether you include, you know, θ_0 or not, in practice, make very little difference to the results. But by convention, usually, we regularize only θ_1 through θ_{100} .

Regularization.

Small values for parameters $\theta_0, \theta_1, \dots, \theta_n$

- "Simpler" hypothesis
- Less prone to overfitting

$$\theta_3, \theta_4 \approx 0$$

Housing:

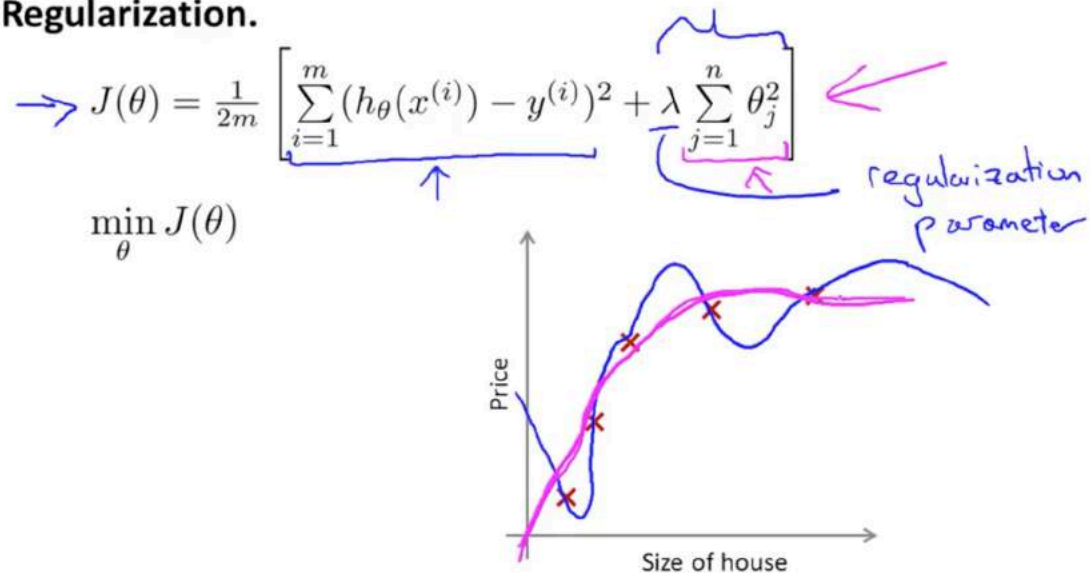
- Features: x_1, x_2, \dots, x_{100}
- Parameters: $\theta_0, \theta_1, \theta_2, \dots, \theta_{100}$

$$J(\theta) = \frac{1}{2m} \left[\sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^n \theta_j^2 \right]$$

~~$\theta_0, \theta_1, \theta_2, \dots, \theta_{100}$~~

Writing down our regularized optimization objective, our regularized cost function again. Here it is. Here's J of θ where, this term on the right is a regularization term and λ here is called the regularization parameter and what λ does, is it controls a trade off between two different goals. The first goal, capture it by the first goal 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 λ , the regularization parameter does is the controls the trade of between these two goals, between the goal of fitting the training set well and the goal of keeping the parameter plan 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 yourselves with regularization you will be able to see this effect firsthand.

Regularization.



Question

In regularized linear regression, we choose θ 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 λ is set to an extremely large value (perhaps too large for our problem, say $\lambda = 10^{10}$)?

- ☐ Algorithm works fine; setting λ to be very large can't hurt it.
- ☐ Algorithm fails to eliminate overfitting.
- ☒ Algorithm results in underfitting (fails to fit even the training set).

Correct

- ☐ Gradient descent will fail to converge.

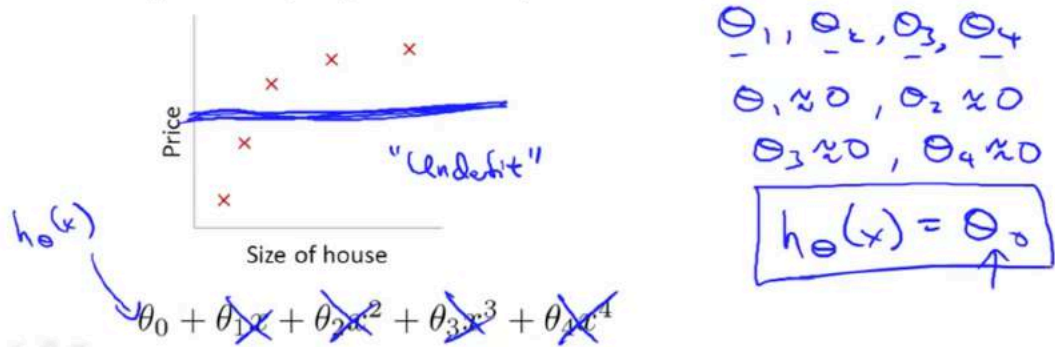
In regularized linear regression, if the regularization parameter λ is set to be very large, then what will happen is we will end up penalizing the parameters $\theta_1, \theta_2, \theta_3, \theta_4$ very highly. That is, if our hypothesis is this is one down at the bottom. And if we end up penalizing $\theta_1, \theta_2, \theta_3, \theta_4$ very heavily, then we end up with all of these parameters close to zero, right? θ_1 will be close to zero; θ_2 will be close to zero. θ_3 and θ_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 zero, 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 fat 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 zero, 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.

In regularized linear regression, we choose θ 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 λ is set to an extremely large value (perhaps for too large for our problem, say $\lambda = 10^{10}$)?



So, that's the idea of the high regularization and the cost function reviews in order to use regularization In the next two videos, lets take these ideas and apply them to linear regression and to logistic regression, so that we can then get them to avoid overfitting.

Cost Function (Transcript)

Note: [5:18 - There is a mini typo. It should be $\sum_{j=1}^n \theta_j^2$ instead of $\sum_{i=1}^n \theta_j^2$]

If we have overfitting from our hypothesis function, we can reduce the weight that some of the terms in our function carry by increasing their cost.

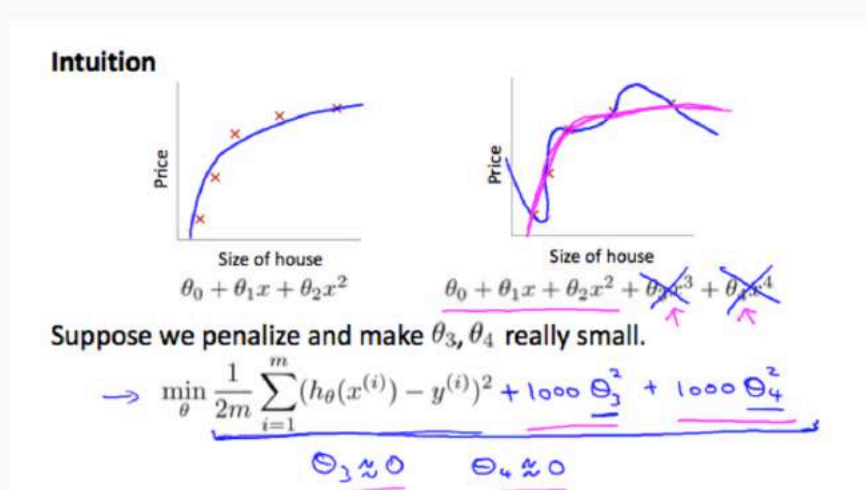
Say we wanted to make the following function more quadratic:

$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

We'll want to eliminate the influence of $\theta_3 x^3$ and $\theta_4 x^4$. Without actually getting rid of these features or changing the form of our hypothesis, we can instead modify our **cost function**:

$$\min_{\theta} \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + 1000 \cdot \theta_3^2 + 1000 \cdot \theta_4^2$$

We've added two extra terms at the end to inflate the cost of θ_3 and θ_4 . Now, in order for the cost function to get close to zero, we will have to reduce the values of θ_3 and θ_4 to near zero. This will in turn greatly reduce the values of $\theta_3 x^3$ and $\theta_4 x^4$ in our hypothesis function. As a result, we see that the new hypothesis (depicted by the pink curve) looks like a quadratic function but fits the data better due to the extra small terms $\theta_3 x^3$ and $\theta_4 x^4$.



We could also regularize all of our theta parameters in a single summation as:

$$\min_{\theta} \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^n \theta_j^2$$

The λ , or lambda, is the **regularization parameter**. It determines how much the costs of our theta parameters are inflated.

Using the above cost function with the extra summation, we can smooth the output of our hypothesis function to reduce overfitting. If lambda is chosen to be too large, it may smooth out the function too much and cause underfitting. Hence, what would happen if $\lambda = 0$ or is too small?

Regularized Linear Regression

(Video)

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 of theta.

Regularized linear regression

$$J(\theta) = \frac{1}{2m} \left[\sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^n \theta_j^2 \right]$$

$\min_{\theta} J(\theta)$

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. 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. 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 zero slightly differently. Concretely, if we want to take this algorithm and modify it to use the regular rise 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 of θ . 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 of θ using the new definition of J of θ with the regularization term. And similarly, this term up on top which I'm drawing the cyan box, that's still the partial derivative respect of θ zero of J of θ . If you look at the update for θ_j , it's possible to show something very interesting. Concretely, θ_j gets updated as θ_j minus α times, and then you have this other term here that depends on θ_j . So if you group all the terms together that depend on θ_j , you can show that this update can be written equivalently as follows. And all I did was add θ_j here is, so θ_j times 1. And this term is, right, λ over m , there's also an α here, so you end up with $\alpha \lambda$ over m multiplied into θ_j . And this term here, $1 - \alpha \lambda$ over m , is a pretty interesting term. It has a pretty interesting effect. Concretely this term, $1 - \alpha \lambda$ over m , is going to be a number that is usually a little bit less than one, because $\alpha \lambda$ 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 θ_j is, we're going to say that θ_j gets replaced by θ_j times 0.99, right? So θ_j times 0.99 has the effect of shrinking θ_j a little bit towards zero. So this makes θ_j a bit smaller. And more formally, this makes the square norm of θ_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 θ_j by a number that's a little bit less than one, so it's 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 of θ that we defined on the previous slide that uses the regularization term.

Gradient descent

Repeat {

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

$$\rightarrow \theta_j := \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \theta_j \right]$$

(j = 1, 2, 3, ..., n)

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

Gradient descent

Repeat {

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

$$\rightarrow \theta_j := \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \theta_j \right]$$

(j = 1, 2, 3, ..., n)

}

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

$1 - \alpha \frac{\lambda}{m} < 1$ 0.99 $\theta_j \times 0.99$ $J(\theta)$ θ_j^2

Question

Suppose you are doing gradient descent on a training set of $m > 0$ examples, using a fairly small learning rate $\alpha > 0$ and some regularization parameter $\lambda > 0$. Consider the update rule:

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

Which of the following statements about the term $(1 - \alpha \frac{\lambda}{m})$ must be true?

- ☐ $1 - \alpha \frac{\lambda}{m} > 1$
- ☐ $1 - \alpha \frac{\lambda}{m} = 1$
- ☒ $1 - \alpha \frac{\lambda}{m} < 1$

Correct

- ☐ None of the above.

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 θ to be equal to this. Right, you have $X^T X$, inverse, $X^T Y$. I'm leaving room here to fill in stuff of course. And what this value for θ does is this minimizes the cost function J of θ , 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. 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 an 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 of θ , with the regularization objective, then this new formula for θ is the one that we give you, the global minimum of J of θ .


Normal equation

$$\begin{aligned}
 \underline{X} &= \begin{bmatrix} (x^{(1)})^T \\ \vdots \\ (x^{(m)})^T \end{bmatrix} \quad \leftarrow \text{m} \times (\text{n}+1) \\
 \underline{y} &= \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix} \quad \mathbb{R}^m \\
 &\rightarrow \min_{\theta} J(\theta) \\
 &\rightarrow \Theta = \left(X^T X + \lambda \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} \right)^{-1} X^T y \\
 &\quad \text{e.g. } n=2 \quad \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (n+1) \times (n+1)
 \end{aligned}$$

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 positive 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, $X^T X$ 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 $X^T X$. Then in this setting, you find that $X^T X$ 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, $X^T X$ 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 $X^T X$ matrix as well.

Non-invertibility (optional/advanced).

Suppose $m \leq n$, 
 (#examples) (#features)

$$\theta = \underbrace{(X^T X)^{-1}}_{\text{non-invertible / singular}} X^T y \quad \begin{array}{cc} \text{pinv} & \text{inv} \\ \text{---} & \text{---} \\ & \nearrow \end{array}$$

If $\lambda > 0$,

$$\theta = \underbrace{\left(X^T X + \lambda \begin{bmatrix} 0 & & & \\ & 1 & & \\ & & 1 & \\ & & & \ddots \\ & & & & 1 \end{bmatrix} \right)^{-1}}_{\text{invertible.}} X^T y$$

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.

Regularized Linear Regression (Transcript)

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 θ_0 from the rest of the parameters because we do not want to penalize θ_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 $\frac{\lambda}{m} \theta_j$ 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, $1 - \alpha \frac{\lambda}{m}$ will always be less than 1. Intuitively you can see it as reducing the value of θ_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 $L = \begin{bmatrix} 0 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix}$

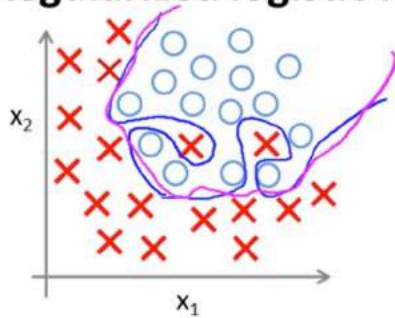
L is a matrix with 0 at the top left and 1's down the diagonal, with 0's everywhere else. It should have dimension $(n+1) \times (n+1)$. Intuitively, this is the identity matrix (though we are not including x_0), multiplied with a single real number λ .

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 (Video)

For logistic regression, we previously talked about two types of optimization algorithms. We talked about how to use gradient descent to optimize a cost function J of θ . And we also talked about advanced optimization methods. Ones that require that you provide a way to compute your cost function J of θ 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 end up with a hypothesis, you know, whose decision bound to be just 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. 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 λ over $2M$, sum from j equals 1, and as usual sum from j equals 1. Rather than the sum from j equals 0, of θ_j squared. And this has the effect therefore, of penalizing the parameters θ_1 , θ_2 and so on up to θ_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.

Regularized logistic regression.



$$h_{\theta}(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^2 x_2 + \theta_4 x_1^2 x_2^2 + \theta_5 x_1^2 x_2^3 + \dots)$$

Cost function:

$$\rightarrow J(\theta) = - \left[\frac{1}{m} \sum_{i=1}^m y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_{\theta}(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

$\theta_1, \theta_2, \dots, \theta_n$

Andrew N

How do we actually implement this? Well, for the original gradient descent algorithm, this was the update we had. 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. 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 of theta. Where J of 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.

Gradient descent

Repeat {

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

$$\rightarrow \theta_j := \theta_j - \alpha \left[\underbrace{\frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}}_{\substack{(j = \text{red } 1, 2, 3, \dots, n) \\ \theta_1, \dots, \theta_n}} + \frac{\lambda}{m} \theta_j \right] \leftarrow$$

}

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

$$h_{\theta}(x) = \frac{1}{1 + e^{-\theta^T x}}$$

Question

When using regularized logistic regression, which of these is the best way to monitor whether gradient descent is working correctly?

- ☐ Plot $-\left[\frac{1}{m} \sum_{i=1}^m y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))\right]$ as a function of the number of iterations and make sure it's decreasing.
- ☐ Plot $-\left[\frac{1}{m} \sum_{i=1}^m y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))\right] - \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$ as a function of the number of iterations and make sure it's decreasing.
- ☒ Plot $-\left[\frac{1}{m} \sum_{i=1}^m y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))\right] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$ as a function of the number of iterations and make sure it's decreasing.
- ☐ Plot $\sum_{j=1}^n \theta_j^2$ as a function of the number of iterations and make sure it's decreasing.

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 as 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 plus 1. And what we needed to do was provide a function. Let's provide a function called cost function that we would then pass in to what we have, what we saw earlier. We will use the fminunc and then you know at

cost function, and so on, right. But the F_{min} , u and 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 J -val. And for that, we need to write code to compute the cost function J of θ . Now, when we're using regularized logistic regression, of course the cost function j of θ 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 of θ be sure to include that term at the end. And then, the other thing that this cost function thing needs to derive with a gradient. So gradient one needs to be set to the partial derivative of J of θ with respect to θ_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 Octave users. 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 θ_0 doesn't change. Compared to the version without regularization. And the other terms do change. And in particular the derivative respect to θ_1 . We worked this out on the previous slide as well. Is equal to, you know, the original term and then minus λ times θ_1 . Just so we make sure we pass this correctly. And 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 of θ . And the parameters you get out will be the ones that correspond to logistic regression with regularization.

Advanced optimization

$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$ $\leftarrow \begin{matrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{matrix}$ $\leftarrow \begin{matrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{matrix}$

\rightarrow `function [jVal, gradient] = costFunction(theta)`

`jVal = [code to compute $J(\theta)$];`

$\rightarrow J(\theta) = \left[-\frac{1}{m} \sum_{i=1}^m y^{(i)} \log(h_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)})) \right] + \left[\frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2 \right]$

\rightarrow `gradient(1) = [code to compute $\frac{\partial}{\partial \theta_0} J(\theta)$];`

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

\rightarrow `gradient(2) = [code to compute $\frac{\partial}{\partial \theta_1} J(\theta)$];`

$\left(\frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_1^{(i)} \right) + \frac{\lambda}{m} \theta_1 \leftarrow$

\rightarrow `gradient(3) = [code to compute $\frac{\partial}{\partial \theta_2} J(\theta)$];`

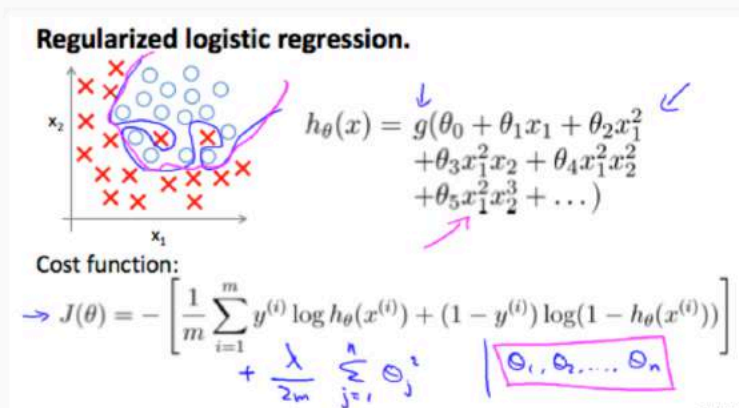
$\vdots \left(\frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_2^{(i)} \right) + \frac{\lambda}{m} \theta_2$

`gradient(n+1) = [code to compute $\frac{\partial}{\partial \theta_n} J(\theta)$];`

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, the advanced optimization algorithms and regularization, by now, frankly, you probably know quite a lot more machine learning than many, certainly now, 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 cause 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 quantifiers that can then 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.

Regularized Logistic Regression (Transcript)

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:



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, θ_0 . I.e. the θ vector is indexed from 0 to n (holding $n+1$ values, θ_0 through θ_n), and this sum explicitly skips θ_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 {

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

$$\rightarrow \theta_j := \theta_j - \alpha \left[\underbrace{\frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}}_{\substack{j = 1, 2, 3, \dots, n \\ \theta_1, \dots, \theta_n}} + \frac{\lambda}{m} \theta_j \right] \leftarrow$$

$\frac{\partial}{\partial \theta_j} J(\theta)$ $h_{\theta}(x) = \frac{1}{1 + e^{-\theta^T x}}$

Quiz

1
point

1. You are training a classification model with logistic regression. Which of the following statements are true? Check all that apply.

- ☐ Adding many new features to the model helps prevent overfitting on the training set.
- ☐ Introducing regularization to the model always results in equal or better performance on the training set.
- ☐ Introducing regularization to the model always results in equal or better performance on examples not in the training set.
- ☒ Adding a new feature to the model always results in equal or better performance on the training set.

1
point

2. Suppose you ran logistic regression twice, once with $\lambda = 0$, and once with $\lambda = 1$. One of the times, you got

parameters $\theta = \begin{bmatrix} 26.29 \\ 65.41 \end{bmatrix}$, and the other time you got

$\theta = \begin{bmatrix} 2.75 \\ 1.32 \end{bmatrix}$. However, you forgot which value of

λ corresponds to which value of θ . Which one do you

think corresponds to $\lambda = 1$?

☒ $\theta = \begin{bmatrix} 2.75 \\ 1.32 \end{bmatrix}$

☐ $\theta = \begin{bmatrix} 26.29 \\ 65.41 \end{bmatrix}$

1
point

3. Which of the following statements about regularization are

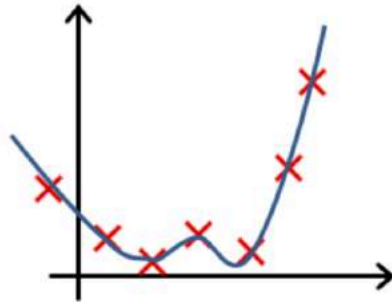
true? Check all that apply.

- ☐ Because regularization causes $J(\theta)$ to no longer be convex, gradient descent may not always converge to the global minimum (when $\lambda > 0$, and when using an appropriate learning rate α).
- ☒ Using too large a value of λ can cause your hypothesis to underfit the data.
- ☐ Using a very large value of λ cannot hurt the performance of your hypothesis; the only reason we do not set λ to be too large is to avoid numerical problems.
- ☐ Because logistic regression outputs values $0 \leq h_{\theta}(x) \leq 1$, its range of output values can only be "shrunk" slightly by regularization anyway, so regularization is generally not helpful for it.

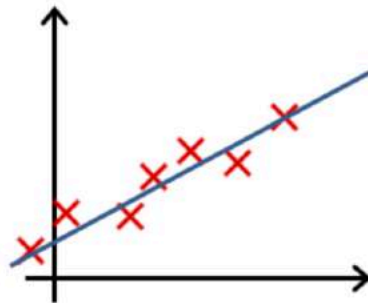
1
point

4. In which one of the following figures do you think the hypothesis has overfit the training set?

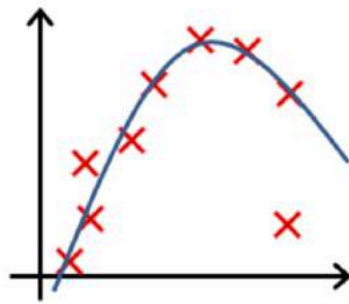
☒ Figure:



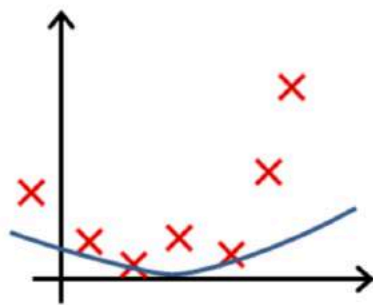
☐ Figure:



☐ Figure:



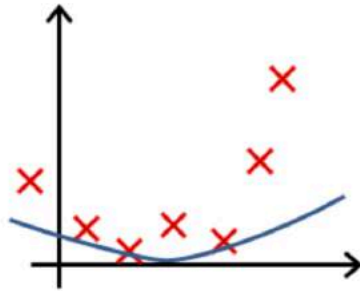
☐ Figure:



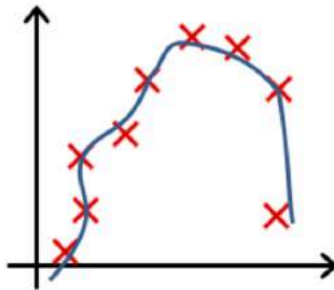
1
point

5. In which one of the following figures do you think the hypothesis has underfit the training set?

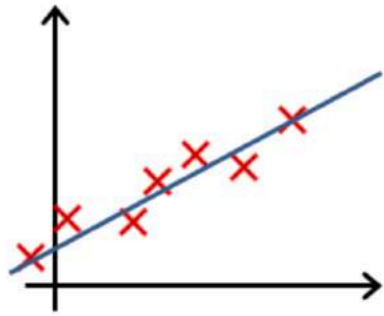
☒ Figure:



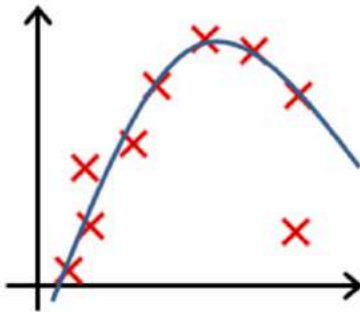
☐ Figure:



☐ Figure:



☐ Figure:



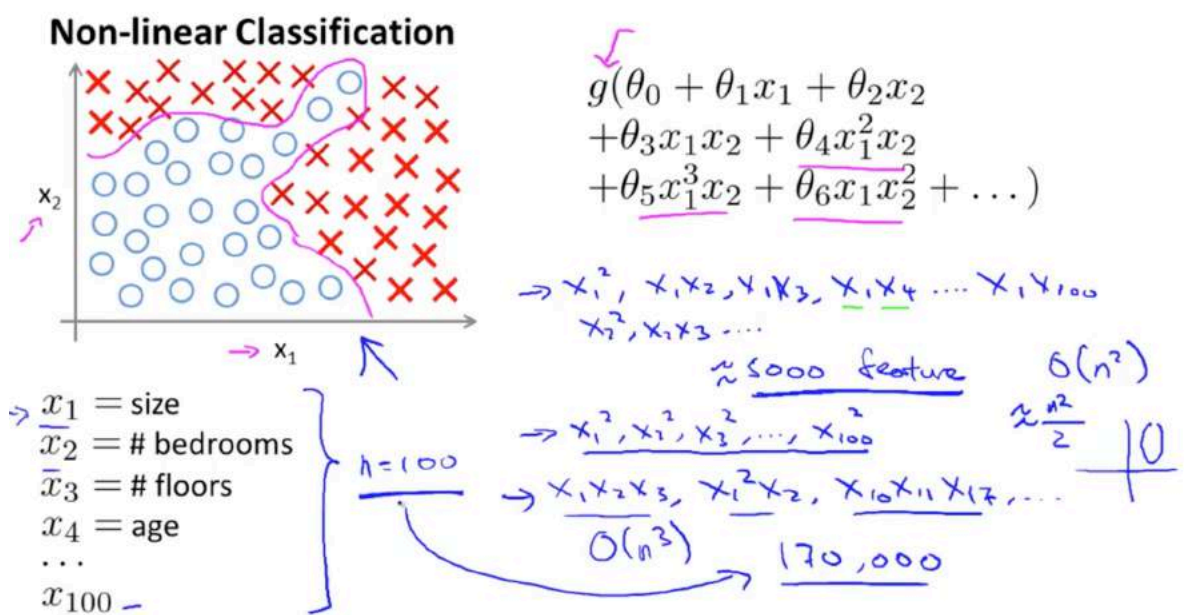
WEEK 4

Motivations

Non-Linear Hypothesis

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. Neural 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 hypothesis 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 or the 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 it's 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 up overfitting the training set, and it can also be computationally

expensive, you know, 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 can 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 known of each others, 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 and cube such features and if any is 100 you can compute that, you end up with on the order of about 170,000 such cubic features and so including these higher auto-polynomial features when your original feature set end 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 none many 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. 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.

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	113	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



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 trained 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



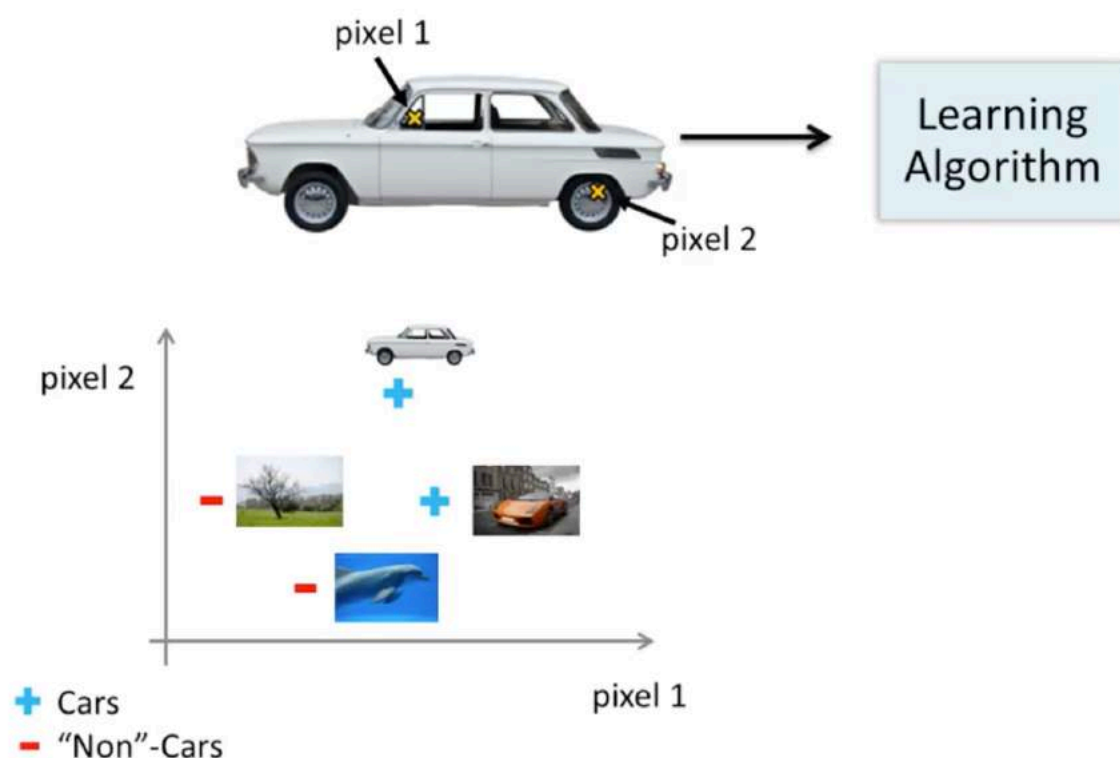
Testing:

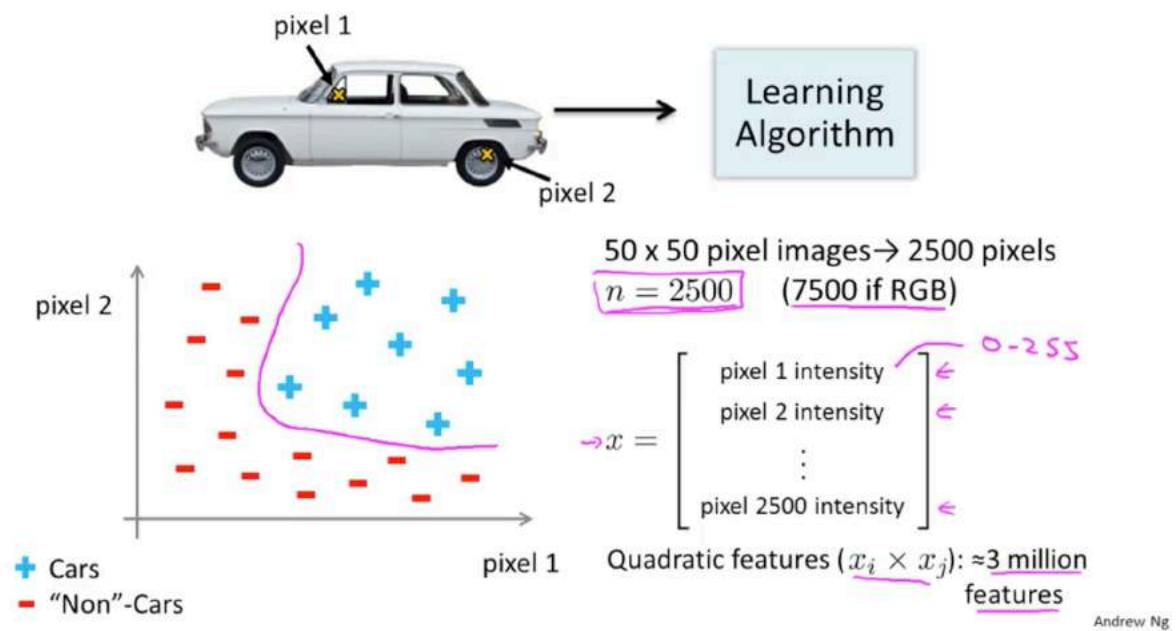


What is this?

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 that 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 equals 2500 where our feature vector x is a list of all the pixel testings, 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 equals 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 equals 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.





Question

Suppose you are learning to recognize cars from 100×100 pixel images (grayscale, not RGB). Let the features be pixel intensity values. If you train logistic regression including all the quadratic terms ($x_i x_j$) as features, about how many features will you have?

- ☐ 5,000
- ☐ 100,000
- ☒ 50 million (5×10^7)

Correct

- ☐ 5 billion (5×10^9)

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. 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.

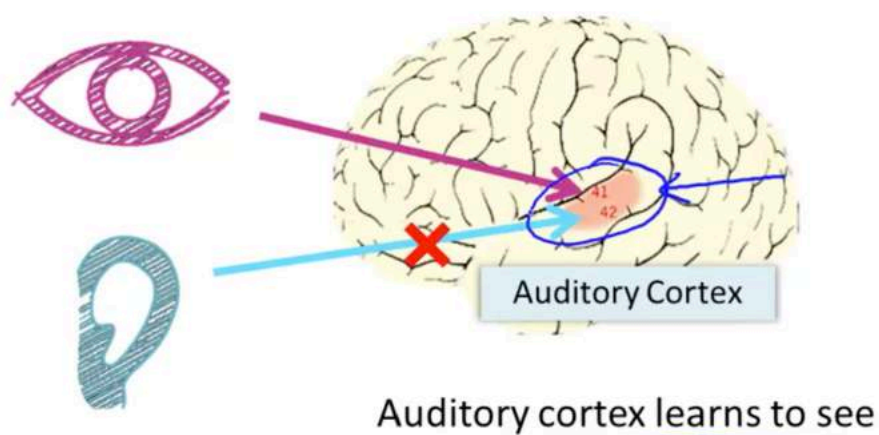
Neural Networks

- **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**

So, when you think about mimicking the brain while one of the human brain does tell me same things, right? The brain can learn to see process images than 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 tell us, but does 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 it is worth just a single learning algorithm. 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

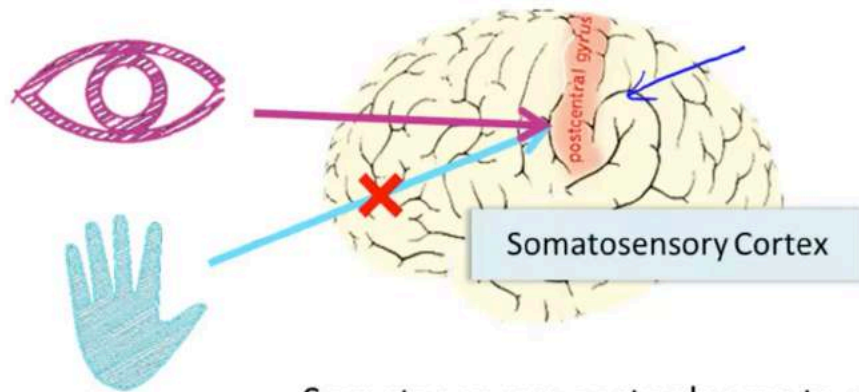
now is your ear is taking the sound signal and routing the sound signal to your auditory cortex and that's what's allowing you to understand my words. Neuroscientists have done the following fascinating experiments where you cut the wire from the ears to the auditory cortex and you re-wire, in this case 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.

The “one learning algorithm” hypothesis



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. 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.

The “one learning algorithm” hypothesis



Somatosensory cortex learns to see

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.

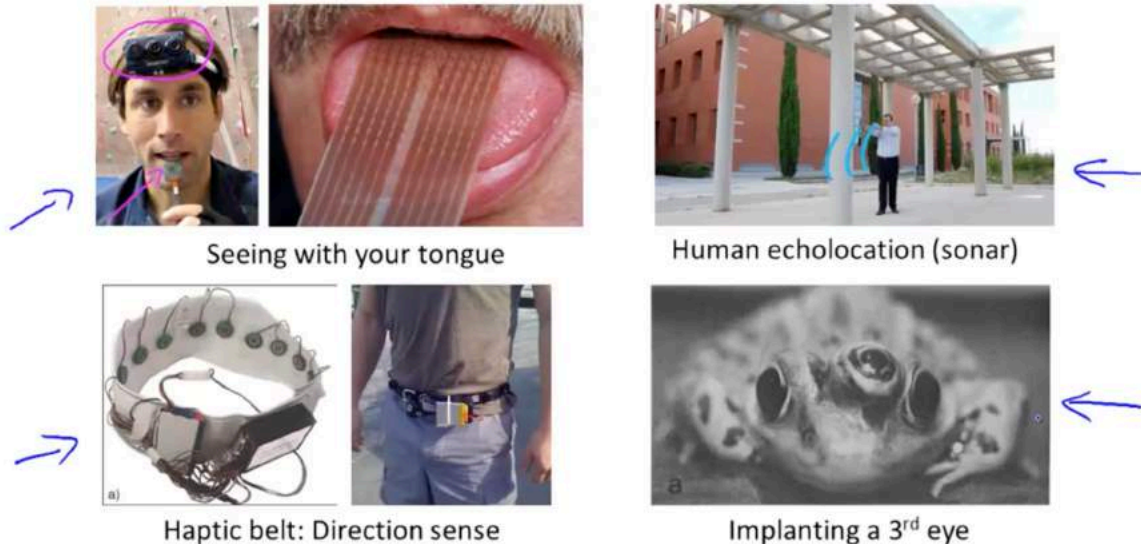
Here's a second example of human echo location or human sonar. So there are two ways you can do this. You can either snap your fingers, or click your tongue. I can't do that very well. But there are blind people today that are actually being trained in schools to do this and learn to interpret the pattern 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 northmost 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.

Sensor representations in the brain



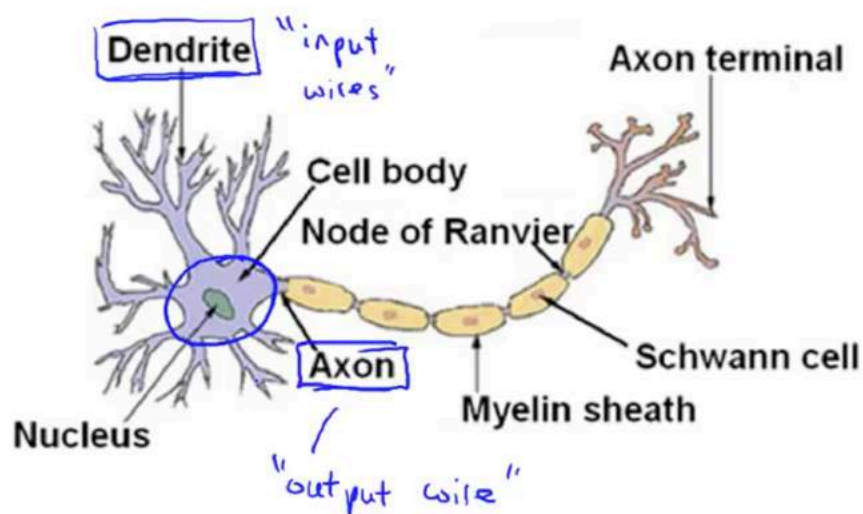
Now, of course, I'm not teaching Neural Networks, you know, just because they might give us a window into this far-off AI dream, even though I'm personally, that's one of the things that I personally work on in my research life. But the main reason I'm teaching Neural Networks in this class is because 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 1 (Video)

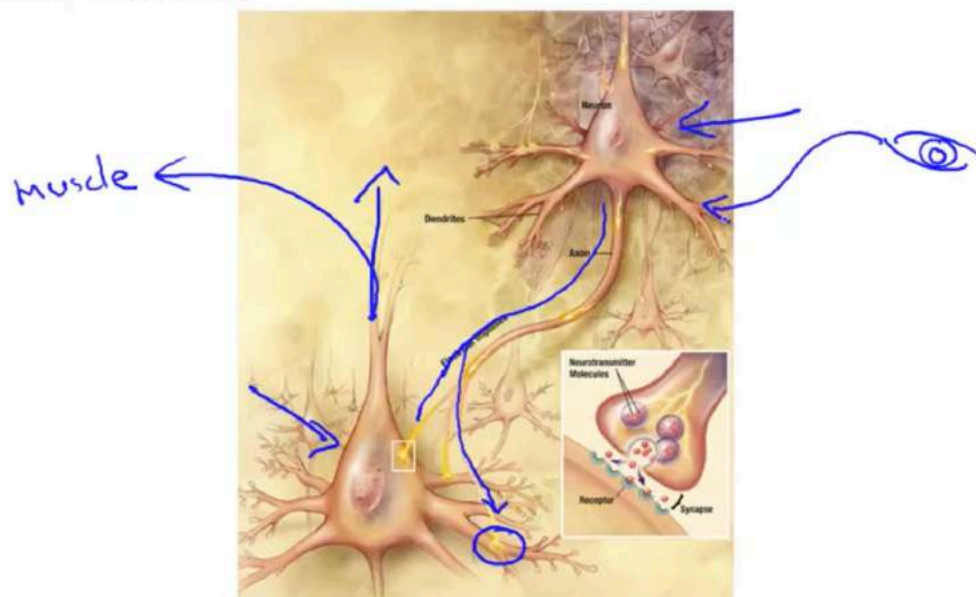
In this video, I want to start telling you about how we represent neural networks. In other words, how we represent our hypothesis or how we represent our model when using neural networks. Neural networks were developed as simulating neurons or networks of neurons in the brain. So, to explain the hypothesis 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 and does some computation and then it says 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. 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. This axon connects to some different neuron and here, this axon that is this open wire, connects to the dendrites of this second neuron over here, which then accepts this incoming message that some computation. And they, in turn, decide to send out this message on this 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 where else in your neuron may send this electricity to your muscle and that causes your muscles to contract and your eyes, some senses like your eye must send a message to your brain while it does it senses hosts electricity entity to a neuron in your brain like so.

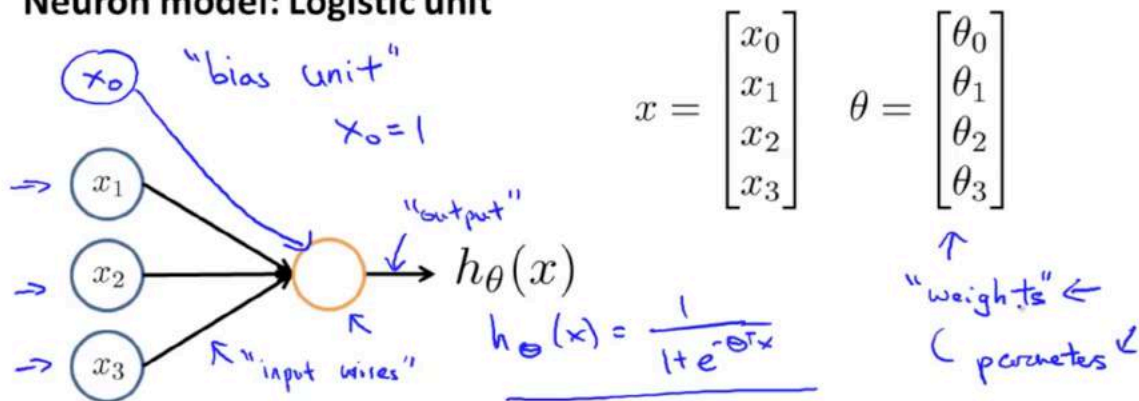
Neurons in the brain



In a neural network, or rather, in an artificial neuron 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 a playing a role analysis, who's maybe the body of a neuron, and we then feed the neuron a few inputs who's various dendrites or input wires. And the neuron does some computation. And output some value on this output wire, or in the biological

neuron, this is an axon. And whenever I draw a diagram like this, what this means is that this represents a computation of h of x equals one over one plus e to the negative theta transpose x , where as usual, x and θ are our parameter vectors, like so. 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. 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 $-z$. And whereas so far I've been calling θ the parameters of the model, I'll mostly continue to use that terminology. Here, it's a copy to the parameters, but in neural networks, in the neural network literature sometimes you might hear people talk about weights of a model and 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.

Neuron model: Logistic unit



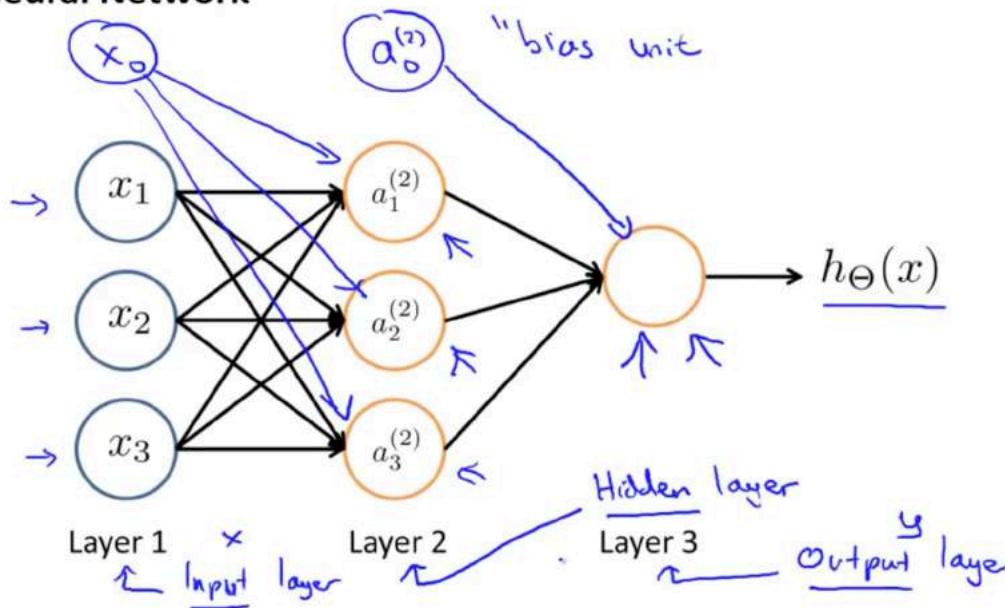
Sigmoid (logistic) activation function.

$$g(z) = \frac{1}{1 + e^{-z}}$$

What a neural network is, is just a group of this different neurons strong together. Completely, here we have input units x_1, x_2, 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 have written 81, 82, 83. I'll talk about those indices later. And once again we can if we want add in just a_0 and

add the mixture bias unit there. There's always a value of 1. And then finally we have this third node and the final layer, and there's this third node that outputs the value that the hypothesis $h(x)$ 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, x_1, x_2, x_3 . The final layer is also called the output layer because that layer has a 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 this ideation is that, you know, you supervised early, where you get to see the inputs and get to see the correct outputs, where there's a hidden layer of values you don't get to observe in the training setup. It's not x , and it's not y , and so we call those hidden. And they try to see neural nets 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 and body represented by this diagram.

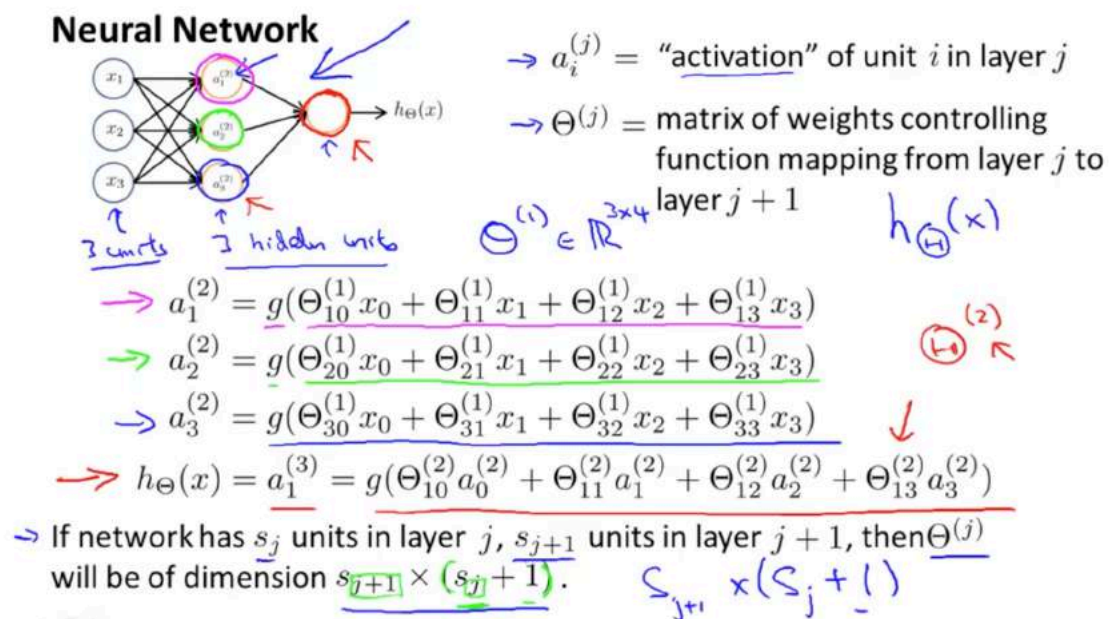
Neural Network



Andrew

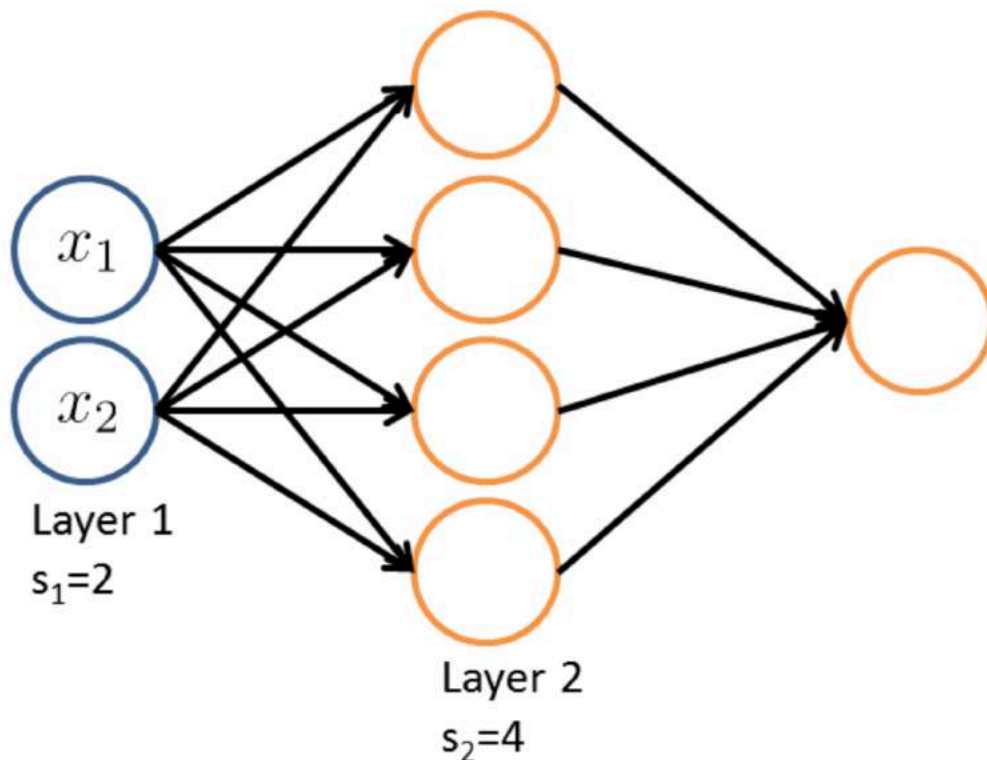
To explain these specific computations represented by a neural network, here's a little bit more notation. I'm going to use a superscript j subscript i to denote the activation of neuron i or of unit i in layer j . So completely this gave superscript to sub group one, that's the activation of the first 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. In addition, new network is parametrize by these matrixes, theta super script j Where 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. So here are the computations that are represented by this diagram. This first hidden unit here has its value computed as follows, there's $a_1^{(2)}$ is equal to the sigma function of the sigma activation function, also called the logistics activation function, apply to this sort of 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 Θ_1 which is matrix of parameters governing our mapping from our three different units, our hidden units. Θ_1 is going to be a 3×4 -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 Θ_j which governs the function mapping from there s_{j+1} . That will have to mention s_{j+1} by $s_j + 1$ I'll just be clear about this notation right. This is 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 $j+1$ 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 loss of this final and after that we have one more unit which computes $h_\Theta(x)$ and that's equal can also be written as $a_1^{(3)}$ and that's equal to this. And you notice that I've written this with a superscript two here, because Θ 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 with x 's input values to hopefully to some space that provisions y . And these hypothesis are parameterized by parameters denoting with a capital Θ so that, as we vary Θ , we get different hypothesis and we get different functions. Mapping say from x to y .



Question

Consider the following neural network:



What is the dimension of $\Theta^{(1)}$?

- ☐ 2×4
- ☐ 4×2
- ☐ 3×4
- ☒ 4×3

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 1

(Transcript)

Let's examine how we will represent a hypothesis function using neural networks. At a very simple level, neurons are basically computational units that take inputs (**dendrites**) as electrical inputs (called "spikes") that are channeled to outputs (**axons**). In our model, our dendrites are like the input features $x_1 \dots x_n$, and the output is the result of our hypothesis function. In this model our x_0 input node is sometimes called the "bias unit." It is always equal to 1. In neural networks, we use the same logistic function as in classification, $\frac{1}{1+e^{-\theta^T x}}$, yet we sometimes call it a sigmoid (logistic) **activation** function. In this situation, our "theta" parameters are sometimes called "weights".

Visually, a simplistic representation looks like:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow [\quad] \rightarrow h_{\theta}(x)$$

Our input nodes (layer 1), also known as the "input layer", go into another node (layer 2), which finally outputs the hypothesis function, known as the "output layer".

We can have intermediate layers of nodes between the input and output layers called the "hidden layers."

In this example, we label these intermediate or "hidden" layer nodes $a_0^2 \dots a_n^2$ and call them "activation units."

$a_i^{(j)}$ = "activation" of unit i in layer j

$\Theta^{(j)}$ = matrix of weights controlling function mapping from layer j to layer $j + 1$

If we had one hidden layer, it would look like:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \rightarrow \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix} \rightarrow h_{\theta}(x)$$

The values for each of the "activation" nodes is obtained as follows:

$$\begin{aligned} a_1^{(2)} &= g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3) \\ a_2^{(2)} &= g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3) \\ a_3^{(2)} &= g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3) \\ h_{\theta}(x) &= a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)}) \end{aligned}$$

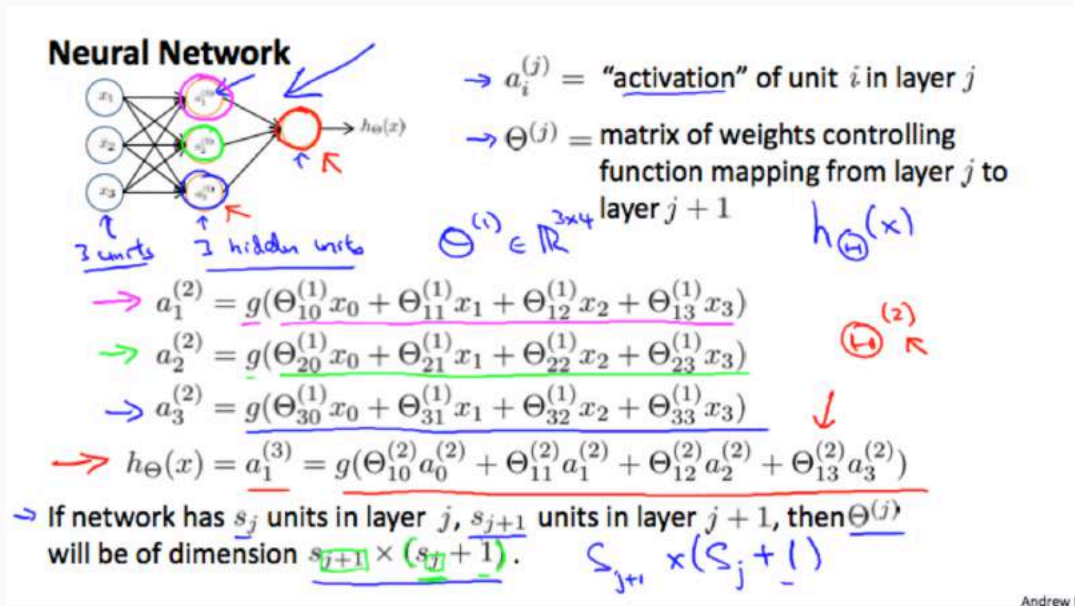
This is saying that we compute our activation nodes by using a 3×4 matrix of parameters. We apply each row of the parameters to our inputs to obtain the value for one activation node. Our hypothesis output is the logistic function applied to the sum of the values of our activation nodes, which have been multiplied by yet another parameter matrix $\Theta^{(2)}$ containing the weights for our second layer of nodes.

Each layer gets its own matrix of weights, $\Theta^{(j)}$.

The dimensions of these matrices of weights is determined as follows:

If network has s_j units in layer j and s_{j+1} units in layer $j + 1$, then $\Theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$.

The +1 comes from the addition in $\Theta^{(j)}$ of the "bias nodes," x_0 and $\Theta_0^{(j)}$. In other words the output nodes will not include the bias nodes while the inputs will. The following image summarizes our model representation:



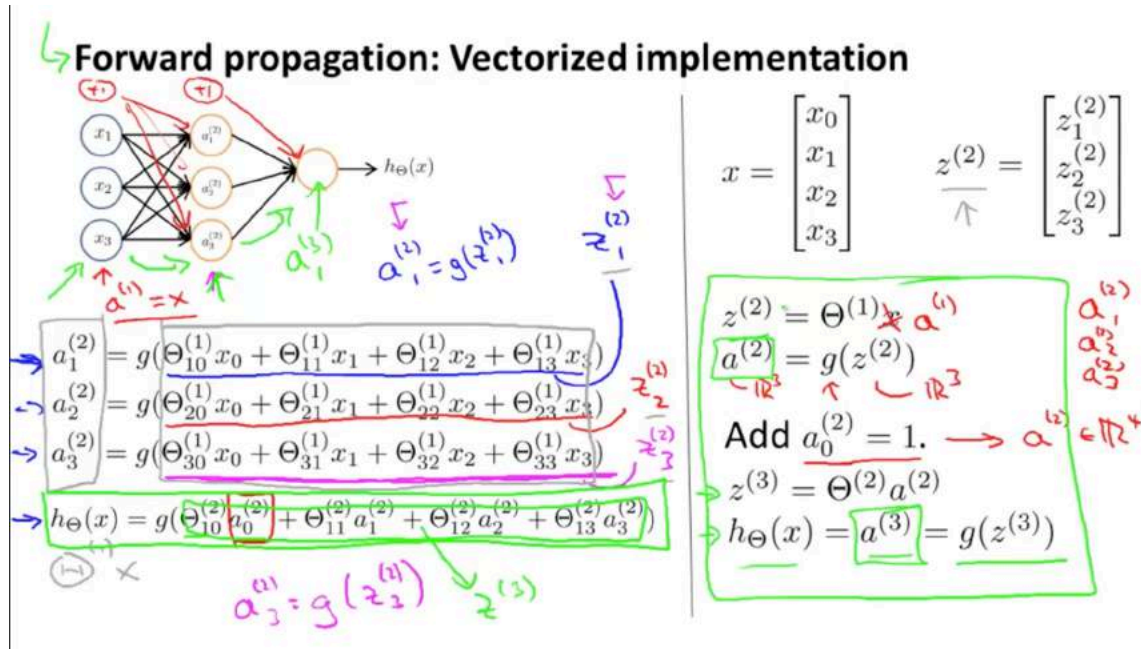
Example: layer 1 has 2 input nodes and layer 2 has 4 activation nodes. Dimension of $\Theta^{(1)}$ is going to be 4×3 where $s_j = 2$ and $s_{j+1} = 4$, so $s_{j+1} \times (s_j + 1) = 4 \times 3$.

Model Representation 2

(Video)

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 vector rise 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 uses and then we use those to compute the final output of our hypotheses h of x . 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 $z^{(2)}_1$. So that we have that $a^{(2)}_1$, which is this term is equal to g of $z^{(2)}_1$. And by the way, these superscript 2, you know, what that means is that the $z^{(2)}$ and this $a^{(2)}$ 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 $z^{(2)}_2$. And finally, this last term here that I'm underlining, let me define that as $z^{(2)}_3$. So that similarly we have $a^{(2)}_3$ equals g of $z^{(2)}_3$. So these z values are just a linear combination, a weighted linear combination, of the input values x_0, x_1, x_2, 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 $x^{(1)}$ times the vector x . Using this observation we're going to be able to vectorize this computation of the neural network. 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 1 and that defines $z^{(2)}$ to be the vector of these z -values, you know, of $z^{(2)}_1, z^{(2)}_2, z^{(2)}_3$. And notice that, there, $z^{(2)}$ this is a three dimensional vector. We can now vectorize the computation of $a^{(2)}_1, a^{(2)}_2, a^{(2)}_3$ as follows. We can just write this in two steps. We can compute $z^{(2)}$ as $\theta^{(2)} x$ and that would give us this vector $z^{(2)}$; and then $a^{(2)}$ is g of $z^{(2)}$ 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 thing it is as in activations of the first layers. So, if I defined $a^{(1)}$ to be equal to x . So, the $a^{(1)}$ is vector, I can now take this x here and replace this with $z^{(1)}$ equals $\theta^{(1)} a^{(1)}$ just by defining $a^{(1)}$ to be activations in my input layer. Now, with what I've written so far I've now gotten myself the values for

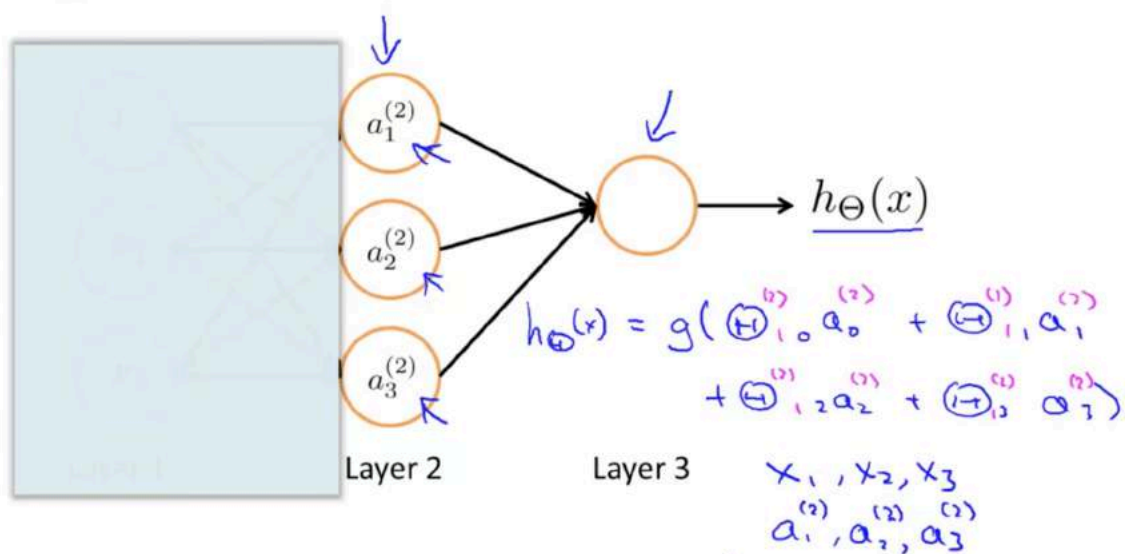
a_1, a_2, a_3 , and really I should put the superscripts there as well. But I need one more value, which is I also want this $a_0^{(2)}$ and that corresponds to a bias unit in the hidden layer that goes to the output there. Of course, there was a bias unit here too that, you know, it just didn't draw 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 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 z_3 . So z_3 is equal to this term here that I'm just underlining. This inner term there is z_3 . And z_3 is stated 2 times a_2 and finally my hypotheses output h of x which is 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 a_3 or as $a_3^{(1)}$ and that's g of z_3 . This process of computing h of x is also called forward propagation and is called that because we start of 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 or both of the efficient way of computing h of x .



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 note, that's just the logistic regression unit and we're using that to make a prediction h of x . And concretely, what the hypotheses is outputting is h of x is going to be equal to g which is my sigmoid activation function times θ_0 times a_0 is equal to 1 plus θ_1 plus θ_2 times a_2 plus θ_3 times a_3 whether values a_1, a_2, a_3 are those given by these three given units. Now, to be actually consistent to my early 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.

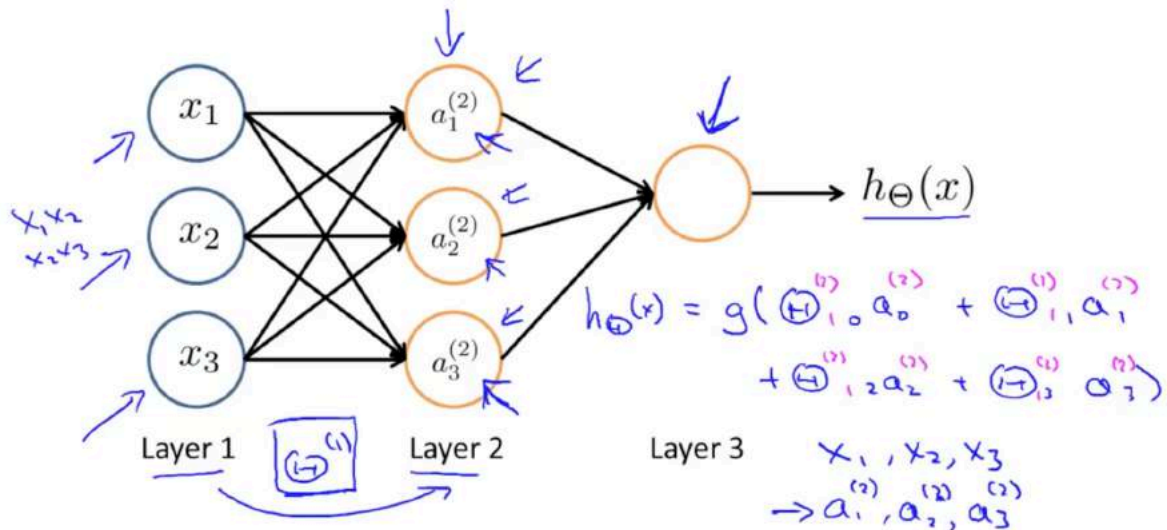
Neural Network learning its own features



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, θ_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 θ_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 at once, using these a_1, a_2, a_3 in order to feed into this last unit that's essentially a logistic regression here.

Neural Network learning its own features



Andrew Ng

I realized this example is described as a 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 there 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 out there working through those examples this explanation will make a little bit more sense.

But just the point. 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. 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 hidden 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.

Other network architectures

