**Gradient Descent**

Θj:=θj−α∑i=1/m(hθ(x(i))−y(i))⋅x(i)j for j := 0...n

**Feature Scaling & Mean Normalization –**

finding z-score for each feature

***If \alphaα is too small:*** slow convergence.

***If \alphaα is too large***: ￼may not decrease on every iteration and thus may not converge.

**Polynomial Regression**

Our hypothesis function need not be linear (a straight line) if that does not fit the data well.

We can change the behavior or curve of our hypothesis function by making it a quadratic, cubic or square root function (or any other form)

**Normal Equation**

In the "Normal Equation" method, we will minimize J by explicitly taking its derivatives with respect to the θj ’s, and setting them to zero. This allows us to find the optimum theta without iteration. The normal equation formula is given below:

θ = (X’ X)-1\*X’ y

*There is no need to do feature scaling with the normal equation.*

The following is a comparison of gradient descent and the normal equation:

|  |  |
| --- | --- |
| Gradient Descent | Normal Equation |
| Need to choose alpha | No need to choose alpha |
| Needs many iterations | No need to iterate |
| Works well when n is large | Slow if n is very large because of (X’X)-1 term |

So if we have a very large number of features, the normal equation will be slow. In practice, when n exceeds 10,000 it might be a good time to go from a normal solution to an iterative process.

**If X^TX is noninvertible, the common causes might be having :**

* Redundant features, where two features are very closely related (i.e. they are linearly dependent)
* Too many features (e.g. m ≤ n). In this case, delete some features or use "regularization" (to be explained in a later lesson).

**Classification**

The classification problem is just like the regression problem, except that the values we now want to predict take on only a small number of discrete values. For now, we will focus on the **binary classification** **problem** in which y can take on only two values, 0 and 1. (Most of what we say here will also generalize to the multiple-class case.) For instance, if we are trying to build a spam classifier for email, then*x*(*i*) may be some features of a piece of email, and y may be 1 if it is a piece of spam mail, and 0 otherwise. Hence, y∈{0,1}. 0 is also called the negative class, and 1 the positive class, and they are sometimes also denoted by the symbols “-” and “+.” Given*x*(*i*), the corresponding *y*(*i*) is also called the label for the training example.

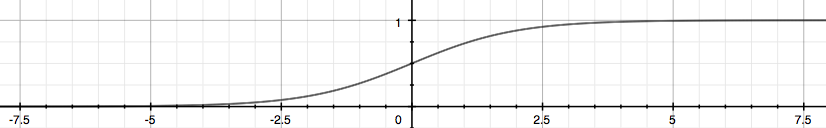
**Logistic Regression**

Let’s change the form for our hypotheses *hθ*​(*x*) to satisfy 0≤*hθ*​(*x*)≤1. This is accomplished by plugging theta=*θTx* into the Logistic Function.

Our new form uses the "Sigmoid Function," also called the "Logistic Function":

|  |
| --- |
| ***hθ*(*x*)=*g*(*θTx)***  ***z*=*θTx***  ***g*(*z*)=1/(1+*e*−*z)*** |

The following image shows us what the sigmoid function looks like:



The function g(z), shown here, maps any real number to the (0, 1) interval, making it useful for transforming an arbitrary-valued function into a function better suited for classification.

*hθ*​(*x*) will give us the **probability** that our output is 1. For example, *hθ*​(*x*)=0.7 gives us a probability of 70% that our output is 1. Our probability that our prediction is 0 is just the complement of our probability that it is 1 (e.g. if probability that it is 1 is 70%, then the probability that it is 0 is 30%).

|  |
| --- |
| ***hθ*(*x*)=*P*(*y*=1|*x*;*θ*)=1−*P*(*y*=0|*x*;*θ*)**  ***P*(*y*=0|*x*;*θ*)+*P*(*y*=1|*x*;*θ*)=1** |

**Decision Boundary**

In order to get our discrete 0 or 1 classification, we can translate the output of the hypothesis function as follows:

|  |
| --- |
| ***hθ*(*x*)≥0.5→*y*=1**  ***hθ*(*x*)<0.5→*y*=0** |

Remember.

|  |
| --- |
| *z*=0,⇒*g*(*z*)=1/2  *z*→∞,*e*∞→0⇒*g*(*z*)=1  *z*→−∞, *e-*∞→∞⇒*g*(*z*)=0 |

So if our input to g is *θTX*, then that means:

|  |
| --- |
| ***hθ*(*x*)=*g*(*θTx*)≥0.5 *whenθTx*≥0** |

From these statements we can now say:

|  |
| --- |
| ***θTx*≥0⇒*y*=1**  ***θTx*<0⇒*y*=0** |

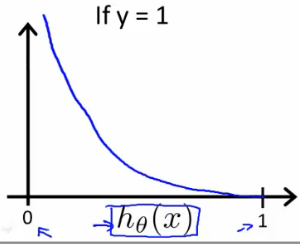
The **decision boundary** is the line that separates the area where y = 0 and where y = 1. It is created by our hypothesis function.

**Cost Function for Logistic Regression**

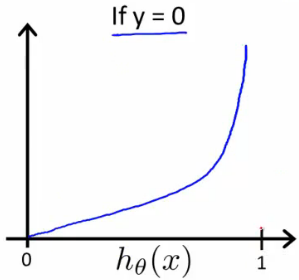
Instead, our cost function for logistic regression looks like:

|  |
| --- |
| ***J*(*θ*)=1/*m*∑*i*=1*m*Cost(*hθ*(*x*(*i*),*y*(*i*))**  **Cost(*hθ*(*x*),*y*)=−log(*hθ*(*x*)) if y = 1**  **Cost(*hθ*(*x*),*y*)=−log(1−*hθ*(*x*)) if y = 0** |
|  |

When y = 1, we get the following plot for *J*(*θ*) vs *hθ*​(*x*):



Similarly, when y = 0, we get the following plot for J(*θ*) vs *hθ*​(*x*):



|  |
| --- |
| **Cost(*hθ*(*x*),*y*)=0 if *hθ*(*x*)=*y***  **Cost(*hθ*(*x*),*y*)→∞ if *y*=0, *hθ*(*x*)→1**  **Cost(*hθ*(*x*),*y*)→∞ if *y*=1and*hθ*(*x*)→0** |

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(θ) is convex for logistic regression.

**Simplified Cost Function and Gradient Descent**

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

**Cost(*hθ*​(*x*),*y*)=−*y*log(*hθ*​(*x*))−(1−*y*)log(1−*hθ*​(*x*))**

Notice that when y is equal to 1, then the second term (1−*y*)log(1−*hθ*​(*x*)) will be zero and will not affect the result. If y is equal to 0, then the first term −*y*log(*hθ*​(*x*)) will be zero and will not affect the result.

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

|  |
| --- |
| ***J*(*θ*)=−1/*m*∑​[*y*(*i*)log(*hθ*​(*x*(*i*)))+(1−*y*(*i*))log(1−*hθ*​(*x*(*i*)))]** |

A vectorized implementation is:

|  |
| --- |
| ***h*=*g*(*Xθ*)**  ***J*(*θ*)=1/*m*⋅(−*yT*log(*h*)−(1−*y*)*T*log(1−*h*))** |

**Gradient Descent**

Remember that the general form of gradient descent is:

|  |
| --- |
| ***Repeat*{**  ***θj*:=*θj*−*α*∂/∂*θj J*(*θ*)**  **}** |

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

|  |
| --- |
| ***Repeat*{**  ***θj*:=*θj*−*α/m*∑(*hθ*(*x*(*i*))−*y*(*i*))*x*(*i*)*j*}** |

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

A vectorized implementation is:

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

# Advanced Optimization

"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*(*θ*)

∂/∂*θj J*(*θ*)

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

**function [jVal, gradient] = costFunction(theta)**

**jVal = [...code to compute J(theta)...];**

**gradient = [...code to compute derivative of J(theta)...];**

**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()".

**options = optimset('GradObj', 'on', 'MaxIter', 100);**

**initialTheta = zeros(2,1);**

**[optTheta, functionVal, exitFlag] = fminunc(@costFunction, initialTheta,**

**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: One-vs-all

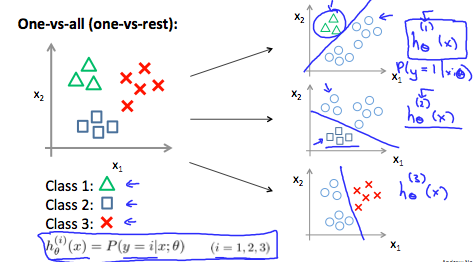
Now we will approach the classification of data when we have more than two categories. Instead of y = {0,1} we will expand our definition so that y = {0,1...n}.

Since y = {0,1...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.

|  |
| --- |
| ***y*∈{0,1...*n*}**  ***h*(0)*θ*(*x*)=*P*(*y*=0|*x*;*θ*)**  ***h*(1)*θ*(*x*)=*P*(*y*=1|*x*;*θ*)**  **.**  **.**  **.**  **h(*n*)*θ*(*x*)=*P*(*y*=*n*|*x*;*θ*)**  **prediction=max*i*(*h*(*i*)*θ*(*x*))** |

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:

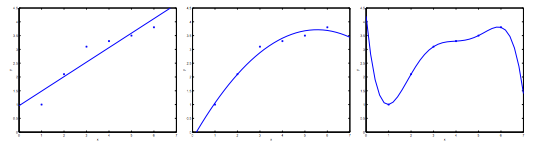


**To summarize:**

To make a prediction on a new x, pick the class that maximizes *hθ*​(*x*)

**The Problem of Overfitting**

Consider the problem of predicting y from x ∈ R. The leftmost figure below shows the result of fitting a y = θ\_0 + θ\_1x*θ*0​+*θ*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*x*2 , and fit y = *y*=*θ*0​+*θ*1​*x*+*θ*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 a5*th* order polynomial *y*=∑*j*=05​*θj*​*xj*. 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 \theta\_j*θj*​.
* Regularization works well when we have a lot of slightly useful features.

# Cost Function for Regularizaation

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.

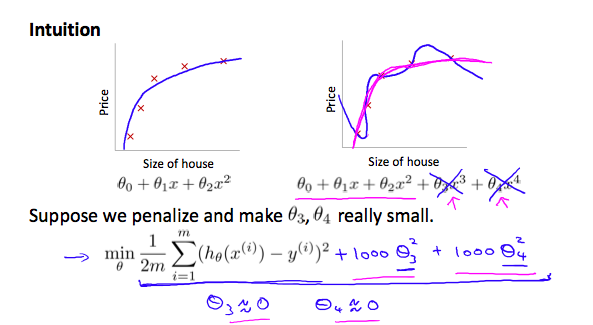
Say we wanted to make the following function more quadratic:

*θ*0​+*θ*1​*x*+*θ*2​*x*2+*θ*3​*x*3+*θ*4​*x*4

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

*minθ*​ ​∑*i*=1*m*​(*hθ*​(*x*(*i*))−*y*(*i*))2+1000⋅*θ*32​+1000⋅*θ*42​

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 *θ*3​*x*3 and *θ*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 *θ*3​*x*3 and *θ*4​*x*4.



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

|  |
| --- |
| ***minθ*​​ ∑*i*=1*m*​(*hθ*​(*x*(*i*))−*y*(*i*))2+*λ* ∑*j*=1*n*​*θj*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 *λ*=0 or is too small ?

# Regularized Linear Regression

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​.

|  |
| --- |
| **Repeat {**  ***θ*0:=*θ*0−*α* ∑(*hθ*(*x*(*i*))−*y*(*i*))*x*(*i*)0**  ***θj*:=*θj*−*α* [(1/*m* ∑(*hθ*(*x*(*i*))−*y*(*i*))*x*(*i*)*j*)+*λ/m θj*]**  **}          *j*∈{1,2...*n*}** |
|  |

The term *mλ*​*θj*​ performs our regularization. With some manipulation our update rule can also be represented as:

***θj*​:=*θj*​(1−*αλ/m*​)−*α/m*​∑ ​(*hθ*​(*x*(*i*))−*y*(*i*))*xj*(*i*)​**

The first term in the above equation,  1−*αmλ*​ will always be less than 1. Intuitively you can see it as reducing the value *θj*​ by some amount on every update. Notice that the second term is now exactly the same as it was before.

### **Normal Equation with Regularization**

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:

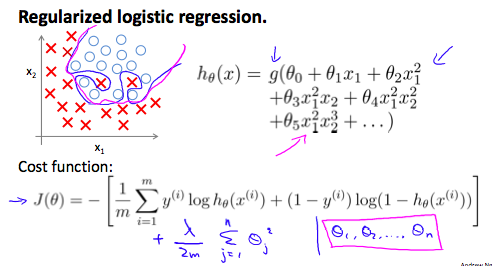
|  |
| --- |
| ***θ*=(*XTX*+*λ*⋅*L*)−1*XTy***  **where  *L*=⎡⎣⎢⎢⎢⎢⎢⎢011⋱1⎤⎦⎥⎥⎥⎥⎥⎥** |

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)×(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 *XTX* is non-invertible. However, when we add the term λ⋅L, then *XTX* + λ⋅L becomes invertible.

# Regularized Logistic Regression

We can regularize logistic regression in a similar way that we regularize linear regression. As a result, we can avoid overfitting. The following image shows how the regularized function, displayed by the pink line, is less likely to overfit than the non-regularized function represented by the blue line:



### Cost Function

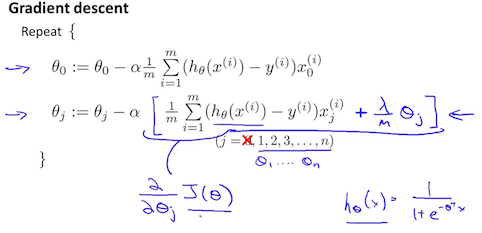
Recall that our cost function for logistic regression was:

***J*(*θ*)=−*m*1​∑*i*=1*m*​[*y*(*i*) log(*hθ*​(*x*(*i*)))+(1−*y*(*i*)) log(1−*hθ*​(*x*(*i*)))]**

We can regularize this equation by adding a term to the end:

|  |
| --- |
| **J(*θ*)=−1/*m*∑ ​[*y*(*i*) log(*hθ*​(*x*(*i*)))+(1−*y*(*i*)) log(1−*hθ*​(*x*(*i*)))]+2*mλ*​∑*j*=1*n*​*θj*2​** |

The second ∑ ​*θj*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 *θn*​), 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:



# Neural Network Model Representation I

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​⋯*xn*​, 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,  **1/1+*e*−*θTx*1**​, 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:

|  |
| --- |
| **[*x*0​*x*1​*x*2​​]→[   ​]→*hθ*​(*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^2\_0 \cdots a^2\_n*a*02​⋯*an*2​ and call them "activation units."

|  |
| --- |
| ***a*(*j*)*i*="activation" of unit *i* in layer *j***  **Θ(*j*)=matrix of weights controlling function mapping from layer *j* to layer *j*+1** |

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

|  |
| --- |
| **[*x*0​*x*1​*x*2​*x*3​​]→[*a*1(2)​*a*2(2)​*a*3(2)​​]→*hθ*​(*x*)** |

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

|  |
| --- |
| ***a*(2)1=*g*(Θ(1)10*x*0+Θ(1)11*x*1+Θ(1)12*x*2+Θ(1)13*x*3)**  ***a*(2)2=*g*(Θ(1)20*x*0+Θ(1)21*x*1+Θ(1)22*x***0**+Θ(1)23*x*3)**  ***a*(2)3=*g*(Θ(1)30*x*0+Θ(1)31*x*1+Θ(1)32*x*2+Θ(1)33*x*3)**  ***h*Θ(*x*)=*a*(3)1=*g*(Θ(2)10*a*(2)0+Θ(2)11*a*(2)1+Θ(2)12*a*(2)2+Θ(2)13*a*(2)3)** |

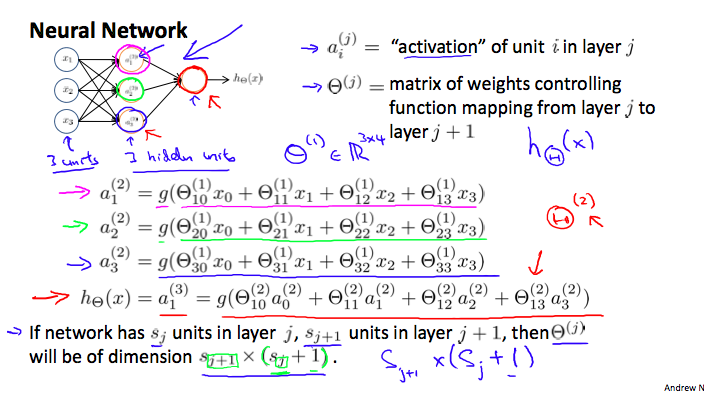
This is saying that we compute our activation nodes by using a 3×4 matrix of parameters. We apply each row of the parameters to our inputs to obtain the value for one activation node. Our hypothesis output is the logistic function applied to the sum of the values of our activation nodes, which have been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

Each layer gets its own matrix of weights, Θ(*j*).

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

**If network has *sj*​ units in layer *j* and *sj*+1​ units in layer *j*+1, then Θ(*j*) will be of dimension *sj*+1​×(*sj*​+1).**

The +1 comes from the addition in Θ(*j*) of the "bias nodes," *x*0​ andΘ0(*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: If layer 1 has 2 input nodes and layer 2 has 4 activation nodes. Dimension of Θ(1) is going to be 4×3 where  *sj*​=2 and  *sj*+1​=4, so  *sj*+1​×(*sj*​+1)=4×3.

# Neural Network Model Representation II

To re-iterate, the following is an example of a neural network:

|  |
| --- |
| ***a*(2)1=*g*(Θ(1)10*x*0+Θ(1)11*x*1+Θ(1)12*x*2+Θ(1)13*x*3)**  ***a*(2)2=*g*(Θ(1)20*x*0+Θ(1)21*x*1+Θ(1)22*x*2+Θ(1)23*x*3**  **)*a*(2)3=*g*(Θ(1)30*x*0+Θ(1)31*x*1+Θ(1)32*x*2+Θ(1)33*x*3)**  ***h*Θ(*x*)=*a*(3)1=*g*(Θ(2)10*a*(2)0+Θ(2)11*a*(2)1+Θ(2)12*a*(2)2+Θ(2)13*a*(2)3)** |

In this section we'll do a vectorized implementation of the above functions. We're going to define a new variable z\_k^{(j)}*zk*(*j*)​ that encompasses the parameters inside our g function. In our previous example if we replaced by the variable z for all the parameters we would get:

|  |
| --- |
| ***a*(2)1=*g*(*z*(2)1)**  ***a*(2)2=*g*(*z*(2)2)**  ***a*(2)3=*g*(*z*(2)3)** |

In other words, for layer j=2 and node k, the variable z will be:

|  |
| --- |
| ***zk*(2)​=Θ*k*,0(1)​*x*0​+Θ*k*,1(1)​*x*1​+⋯+Θ*k*,*n*(1)​*xn*​** |

The vector representation of x and *zj* is:

|  |
| --- |
| ***x*=⎡⎣⎢⎢*x*0*x*1⋯*xn*⎤⎦⎥⎥**  ***z*(*j*)=⎡⎣⎢⎢⎢⎢*z*(*j*)1*z*(*j*)2⋯*z*(*j*)*n*⎤⎦⎥⎥⎥⎥** |

Setting  *x*=*a*(1), we can rewrite the equation as:

|  |
| --- |
| ***z*(*j*)=Θ(*j*−1)*a*(*j*−1)** |

We are multiplying our matrix Θ(*j*−1) with dimensions  *sj*​×(*n*+1) (where *sj*​ is the number of our activation nodes) by our vector  *a*(*j*−1) with height (n+1). This gives us our vector  *z*(*j*) with height *sj*​. Now we can get a vector of our activation nodes for layer j as follows:

***a*(*j*)=*g*(*z*(*j*))**

Where our function g can be applied element-wise to our vector  *z*(*j*).

We can then add a bias unit (equal to 1) to layer j after we have computed  *a*(*j*). This will be element  *a*0(*j*)​ and will be equal to 1. To compute our final hypothesis, let's first compute another z vector:

***z*(*j*+1)=Θ(*j*)*a*(*j*)**

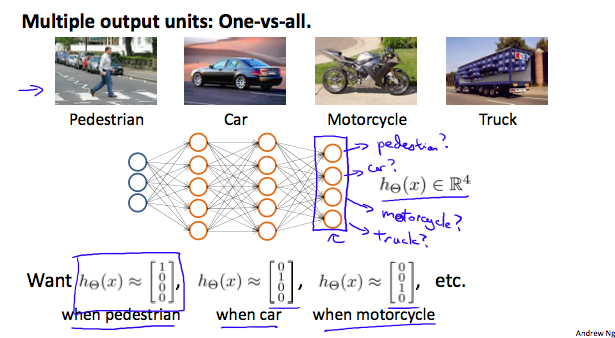
We get this final z vector by multiplying the next theta matrix after \Theta^{(j-1)}Θ(*j*−1) with the values of all the activation nodes we just got. This last theta matrix \Theta^{(j)}Θ(*j*) will have only **one row**which is multiplied by one column a^{(j)}*a*(*j*) so that our result is a single number. We then get our final result with:

***h*Θ​(*x*)=*a*(*j*+1)=*g*(*z*(*j*+1))**

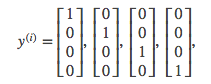
Notice that in this **last step**, between layer j and layer j+1, we are doing **exactly the same thing** as we did in logistic regression. Adding all these intermediate layers in neural networks allows us to more elegantly produce interesting and more complex non-linear hypotheses.

# Neural Network Multiclass Classification

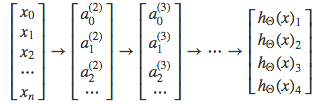
To classify data into multiple classes, we let our hypothesis function return a vector of values. Say we wanted to classify our data into one of four categories. We will use the following example to see how this classification is done. This algorithm takes as input an image and classifies it accordingly:



We can define our set of resulting classes as y:



Each *y*(*i*) represents a different image corresponding to either a car, pedestrian, truck, or motorcycle. The inner layers, each provide us with some new information which leads to our final hypothesis function. The setup looks like:



Our resulting hypothesis for one set of inputs may look like:

|  |
| --- |
| *h*Θ​(*x*)=[0010​] |

In which case our resulting class is the third one down, or *h*Θ​(*x*)3​, which represents the motorcycle.

**Neural Network Cost Function**

Let's first define a few variables that we will need to use:

* L = total number of layers in the network
* *sl*​= number of units (not counting bias unit) in layer l
* K = number of output units/classes

Recall that in neural networks, we may have many output nodes. We denote  *h*Θ​(*x*)*k*​ as being a hypothesis that results in the*kth* output. Our cost function for neural networks is going to be a generalization of the one we used for logistic regression. Recall that the cost function for regularized logistic regression was:

|  |
| --- |
| ***J*(*θ*)=−*m*1​∑*i*=1*m*​[*y*(*i*) log(*hθ*​(*x*(*i*)))+(1−*y*(*i*)) log(1−*hθ*​(*x*(*i*)))]+2*mλ*​∑*j*=1*n*​*θj*2​** |

For neural networks, it is going to be slightly more complicated:

|  |
| --- |
| ***J*(Θ)=−1*m*∑*i*=1*m*∑*k*=1*K*[*y*(*i*)*k*log((*h*Θ(*x*(*i*)))*k*)+(1−*y*(*i*)*k*)log(1−(*h*Θ(*x*(*i*)))*k*)]+*λ*2*m*∑*l*=1*L*−1∑*i*=1*sl*∑*j*=1*sl*+1(Θ(*l*)*j*,*i*)2** |

We have added a few nested summations to account for our multiple output nodes. In the first part of the equation, before the square brackets, we have an additional nested summation that loops through the number of output nodes.

In the regularization part, after the square brackets, we must account for multiple theta matrices. The number of columns in our current theta matrix is equal to the number of nodes in our current layer (including the bias unit). The number of rows in our current theta matrix is equal to the number of nodes in the next layer (excluding the bias unit). As before with logistic regression, we square every term.

Note:

* the double sum simply adds up the logistic regression costs calculated for each cell in the output layer
* the triple sum simply adds up the squares of all the individual Θs in the entire network.
* the i in the triple sum does **not** refer to training example i

**Backpropagation Algorithm**

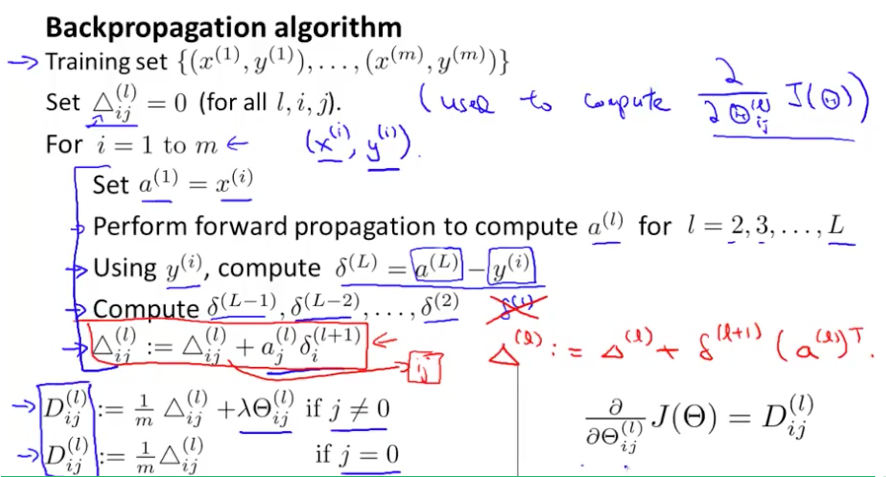
"Backpropagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression. Our goal is to compute:

**minΘ​*J*(Θ**)

That is, we want to minimize our cost function J using an optimal set of parameters in theta. In this section we'll look at the equations we use to compute the partial derivative of J(Θ):

**∂Θ*i*,*j*(*l*)​∂​*J*(Θ)**

To do so, we use the following algorithm:



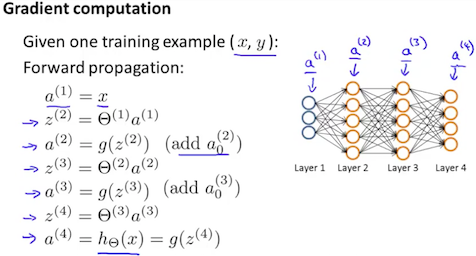
**Back propagatio**

{(*x*(1),*y*(1))⋯(*x*(*m*),*y*(*m*))}

* **Δ*i*,*j*(*l*)​ := 0 for all (l,i,j),** (hence you end up having a matrix full of zeros)

For training example t =1 to m:

1. **Set *a*(1):=*x*(*t*)**
2. **Perform forward propagation to compute a(*l*) for l=2,3,…,L**



**3. Using *y*(*t*), compute *δ*(*L*)=*a*(*L*)−*y*(*t*)**

Where L is our total number of layers and *a*(*L*) is the vector of outputs of the activation units for the last layer. So our "error values" for the last layer are simply the differences of our actual results in the last layer and the correct outputs in y. To get the delta values of the layers before the last layer, we can use an equation that steps us back from right to left:

* 1. **Compute *δ*(*L*−1),*δ*(*L*−2),…,*δ*(2) using**

***δ*(*l*)=((Θ(*l*))*Tδ*(*l*+1)) .∗ *a*(*l*) .∗ (1−*a*(*l*))**

The delta values of layer l are calculated by multiplying the delta values in the next layer with the theta matrix of layer l. We then element-wise multiply that with a function called g', or g-prime, which is the derivative of the activation function g evaluated with the input values given by *z*(*l*).

The g-prime derivative terms can also be written out as:

|  |
| --- |
| ***g*′(*z*(*l*))=*a*(*l*) .∗ (1−*a*(*l*))** |

**5.  Δ(*l*):=Δ(*l*)+*δ*(*l*+1)(*a*(*l*))*T***

Hence we update our new Δ matrix.

* ***Di*,*j*(*l*)​:=*m*1​(Δ*i*,*j*(*l*)​+*λ*Θ*i*,*j*(*l*)​), if j≠0.**

*Di*,*j*(*l*)​:=*m*1​Δ*i*,*j*(*l*)​ If j=0

The capital-delta matrix D is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative. Thus we *Dij*(*l*)​

# Backpropagation Intuition

Recall that the cost function for a neural network is:

|  |
| --- |
| ***J*(Θ)=−1*m*∑*t*=1*m*∑*k*=1*K*[*y*(*t*)*k* log(*h*Θ(*x*(*t*)))*k*+(1−*y*(*t*)*k*) log(1−*h*Θ(*x*(*t*))*k*)]+*λ*2*m*∑*l*=1*L*−1∑*i*=1*sl*∑*j*=1*sl*+1(Θ(*l*)*j*,*i*)2** |

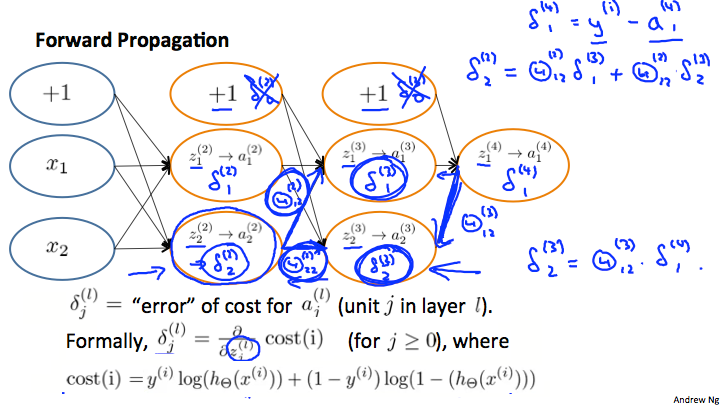
If we consider simple non-multiclass classification (k = 1) and disregard regularization, the cost is computed with:

|  |
| --- |
| ***cost*(*t*)=*y*(*t*) log(*h*Θ​(*x*(*t*)))+(1−*y*(*t*)) log(1−*h*Θ​(*x*(*t*)))** |

Intuitively, *δj*(*l*)​ is the "error" for *aj*(*l*)​ (unit j in layer l). More formally, the delta values are actually the derivative of the cost function:

|  |
| --- |
| **δj(l)​=∂zj(l)​∂​cost(t)** |

Recall that our derivative is the slope of a line tangent to the cost function, so the steeper the slope the more incorrect we are. Let us consider the following neural network below and see how we could calculate some *δj*(*l*)​:



In the image above, to calculate *δ*2(2)​, we multiply the weights Θ12(2)​ and Θ22(2)​ by their respective *δ* values found to the right of each edge. So we get *δ*2(2)​= Θ12(2)​\**δ*1(3)​+ Θ22(2)​\* *δ*2(3)​. To calculate every single possible *δj*(*l*)​, we could start from the right of our diagram. We can think of our edges as our Θ*ij*​. Going from right to left, to calculate the value of *δj*(*l*)​, you can just take the over all sum of each weight times the *δ* it is coming from. Hence, another example would be *δ*2(3)​= Θ12(3)​\* *δ*1(4)​.

# Implementation Note: Unrolling Parameters

With neural networks, we are working with sets of matrices:

|  |
| --- |
| Θ(1),Θ(2),Θ(3),…*D*(1),*D*(2),*D*(3),… |

In order to use optimizing functions such as "fminunc()", we will want to "unroll" all the elements and put them into one long vector:



**thetaVector = [ Theta1(:); Theta2(:); Theta3(:); ]**

**deltaVector = [ D1(:); D2(:); D3(:) ]**

If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11, then we can get back our original matrices from the "unrolled" versions as follows:



3

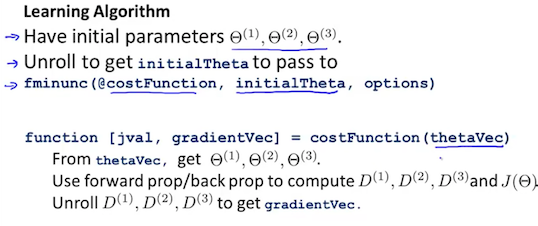
4

**Theta1 = reshape(thetaVector(1:110),10,11)**

**Theta2 = reshape(thetaVector(111:220),10,11)**

**Theta3 = reshape(thetaVector(221:231),1,11)**

To summarize:



# Gradient Checking

Gradient checking will assure that our backpropagation works as intended. We can approximate the derivative of our cost function with:

**∂Θ∂​*J*(Θ)≈2*ϵJ*(Θ+*ϵ*)−*J*(Θ−*ϵ*)**​

With multiple theta matrices, we can approximate the derivative **with respect to** Θ*j*​ as follows:

**∂Θ*j*​∂​*J*(Θ)≈2*ϵJ*(Θ1​,…,Θ*j*​+*ϵ*,…,Θ*n*​)−*J*(Θ1​,…,Θ*j*​−*ϵ*,…,Θ*n*​)​**

A small value for *ϵ* (epsilon) such as *ϵ*=10−4, guarantees that the math works out properly. If the value for epsilon*ϵ* is too small, we can end up with numerical problems.

Hence, we are only adding or subtracting epsilon to the Θ*j*​ matrix. In octave we can do it as follows:



1

2

3

4

5

6

7

8

9

**epsilon = 1e-4;**

**for i = 1:n,**

**thetaPlus = theta;**

**thetaPlus(i) += epsilon;**

**thetaMinus = theta;**

**thetaMinus(i) -= epsilon;**

**gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2\*epsilon)**

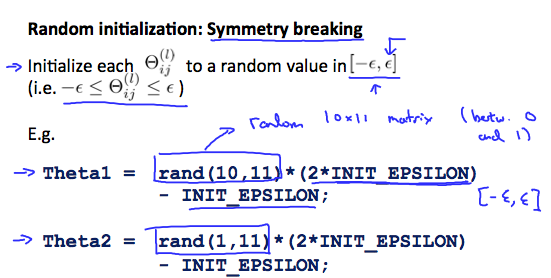
**end;**

We previously saw how to calculate the deltaVector. So once we compute our gradApprox vector, we can check that gradApprox ≈ deltaVector.

**Once you have verified**once**that your backpropagation algorithm is correct, you don't need to compute gradApprox again. The code to compute gradApprox can be very slow.**

# Random Initialization

Initializing all theta weights to zero does not work with neural networks. When we backpropagate, all nodes will update to the same value repeatedly. Instead we can randomly initialize our weights for our Θ matrices using the following method:



Hence, we initialize each Θ*ij*(*l*)​ to a random value between[[−*ϵ*,*ϵ*]. Using the above formula guarantees that we get the desired bound. The same procedure applies to all the Θ's. Below is some working code you could use to experiment.



**If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11.**

**Theta1 = rand(10,11) \* (2 \* INIT\_EPSILON) - INIT\_EPSILON;**

**Theta2 = rand(10,11) \* (2 \* INIT\_EPSILON) - INIT\_EPSILON;**

**Theta3 = rand(1,11) \* (2 \* INIT\_EPSILON) - INIT\_EPSILON;**

rand(x,y) is just a function in octave that will initialize a matrix of random real numbers between 0 and 1.

(Note: the epsilon used above is unrelated to the epsilon from Gradient Checking)

**Putting it Together**

First, pick a network architecture; choose the layout of your neural network, including how many hidden units in each layer and how many layers in total you want to have.

* Number of input units = dimension of features x^{(i)}*x*(*i*)
* Number of output units = number of classes
* Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units)
* Defaults: 1 hidden layer. If you have more than 1 hidden layer, then it is recommended that you have the same number of units in every hidden layer.

**Training a Neural Network**

1. **Randomly initialize the weights**
2. **Implement forward propagation to get h\_\Theta(x^{(i)})*h*Θ​(*x*(*i*)) for any x^{(i)}*x*(*i*)**
3. **Implement the cost function**
4. **Implement backpropagation to compute partial derivatives**
5. **Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.**
6. **Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.**

When we perform forward and back propagation, we loop on every training example:



1

2

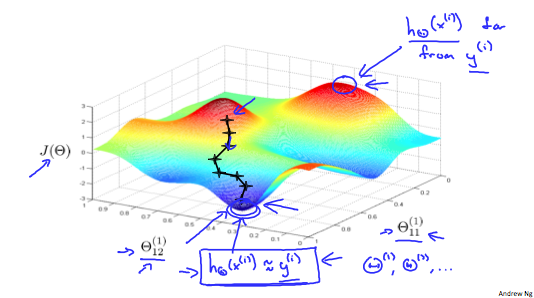
3

for i = 1:m,

Perform forward propagation and backpropagation using example (x(i),y(i))

(Get activations a(l) and delta terms d(l) for l = 2,...,L

The following image gives us an intuition of what is happening as we are implementing our neural network:



Ideally, you want *h*Θ​(*x*(*i*)) approx≈*y*(*i*). This will minimize our cost function. However, keep in mind that*J*(Θ) is not convex and thus we can end up in a local minimum instead.

# Evaluating a Hypothesis

Once we have done some trouble shooting for errors in our predictions by:

* Getting more training examples
* Trying smaller sets of features
* Trying additional features
* Trying polynomial features
* Increasing or decreasing λ

We can move on to evaluate our new hypothesis.

A hypothesis may have a low error for the training examples but still be inaccurate (because of overfitting). Thus, to evaluate a hypothesis, given a dataset of training examples, we can split up the data into two sets: a **training set** and a **test set**. Typically, the training set consists of 70 % of your data and the test set is the remaining 30 %.

The new procedure using these two sets is then:

1. **Learn Θ and minimize *Jtrain*​(Θ) using the training set**
2. **Compute the test set error *Jtest*​(Θ)**

***Jtest*​(Θ)=2*mtest*​1​∑*i*=1*mtest*​​(*h*Θ​(*xtest*(*i*)​)−*ytest*(*i*)​)2**

1. For classification ~ Misclassification error (aka 0/1 misclassification error):

|  |
| --- |
| *err*(*h*Θ(*x*),*y*)=10if *h*Θ(*x*)≥0.5 *and* *y*=0 *or* *h*Θ(*x*)<0.5 *and* *y*=1o*therwise* |

This gives us a binary 0 or 1 error result based on a misclassification. The average test error for the test set is:

**Test Error=*mtest*​1​∑*i*=1*mtest*​​*err*(*h*Θ​(*xtest*(*i*)​),*ytest*(*i*)​)**

This gives us the proportion of the test data that was misclassified.

**Model Selection and Train/Validation/Test Sets**

Just because a learning algorithm fits a training set well, that does not mean it is a good hypothesis. It could over fit and as a result your predictions on the test set would be poor. The error of your hypothesis as measured on the data set with which you trained the parameters will be lower than the error on any other data set.

Given many models with different polynomial degrees, we can use a systematic approach to identify the 'best' function. In order to choose the model of your hypothesis, you can test each degree of polynomial and look at the error result.

One way to break down our dataset into the three sets is:

* **Training set: 60%**
* **Cross validation set: 20%**
* **Test set: 20%**

We can now calculate three separate error values for the three different sets using the following method:

1. Optimize the parameters in Θ using the training set for each polynomial degree.
2. Find the polynomial degree d with the least error using the cross validation set.
3. Estimate the generalization error using the test set with *Jtest*​(Θ(*d*)), (d = theta from polynomial with lower error);

This way, the degree of the polynomial d has not been trained using the test set.

**Diagnosing Bias vs. Variance**

In this section we examine the relationship between the degree of the polynomial d and the underfitting or overfitting of our hypothesis.

* We need to distinguish whether **bias** or **variance** is the problem contributing to bad predictions.
* High bias is underfitting and high variance is overfitting. Ideally, we need to find a golden mean between these two.

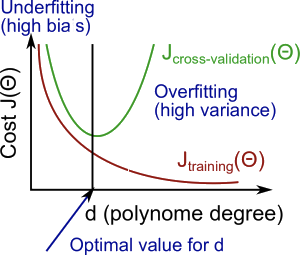
The training error will tend to **decrease** as we increase the degree d of the polynomial.

At the same time, the cross validation error will tend to **decrease** as we increase d up to a point, and then it will **increase** as d is increased, forming a convex curve.

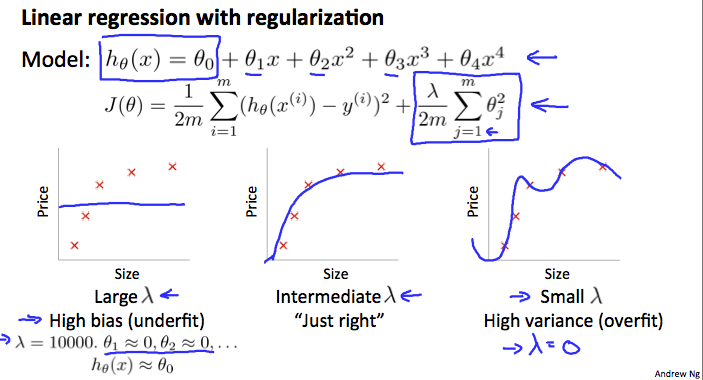
**High bias (underfitting): both *Jtrain*​(Θ) and *JCV*​(Θ) will be high. Also,*JCV*​(Θ)≈*Jtrain*​(Θ).**

**High variance (overfitting):J*train*​(Θ) will be low and *JCV*​(Θ) will be much greater than *Jtrain*​(Θ).**

The is summarized in the figure below:



**Regularization and Bias/Variance**



In the figure above, we see that as *λ* increases, our fit becomes more rigid. On the other hand, as *λ* approaches 0, we tend to over overfit the data. So how do we choose our parameter *λ* to get it 'just right' ? In order to choose the model and the regularization term λ, we need to:

1. **Create a list of lambdas (i.e. λ∈{0,0.01,0.02,0.04,0.08,0.16,0.32,0.64,1.28,2.56,5.12,10.24});**
2. **Create a set of models with different degrees or any other variants.**
3. **Iterate through the *λ*s and for each *λ* go through all the models to learn some Θ.**
4. **Compute the cross validation error using the learned Θ (computed with λ) on the *JCV*​(Θ) without regularization or λ = 0.**
5. **Select the best combo that produces the lowest error on the cross validation set.**
6. **Using the best combo Θ and λ, apply it on*Jtest*​(Θ) to see if it has a good generalization of the problem.**