Week5 Lecture Notes

ML: Neural Networks: Learning

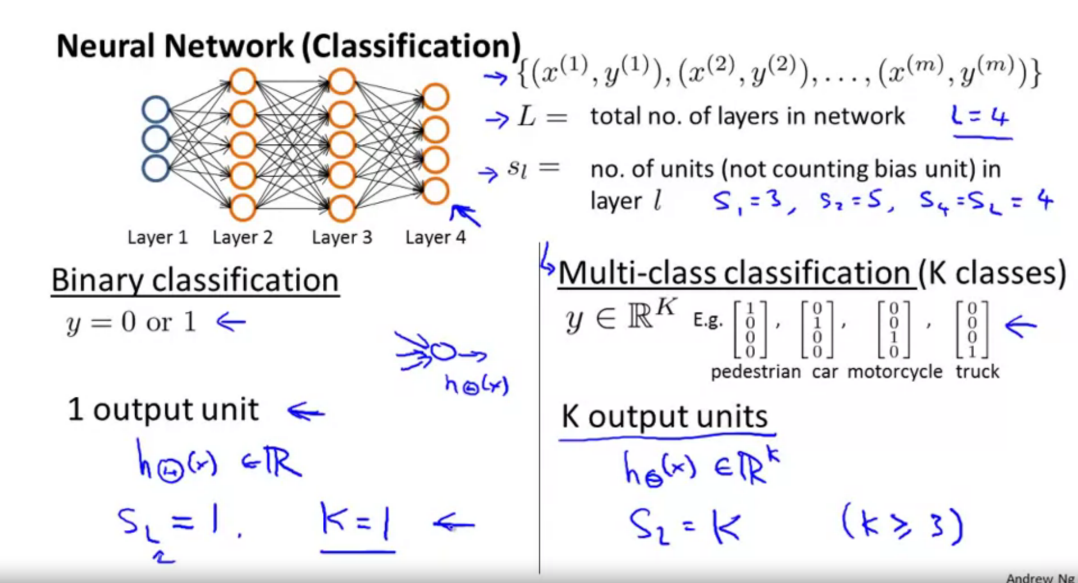
**Cost Function**

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

a) L= total number of layers in the network

b) = number of units (not counting bias unit) in layer l

c) K= number of output units/classes

Recall that in neural networks, we may have many output nodes. We denote as being a hypothesis that results in the 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:

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

We have added a few nested summations to account for our multiple output nodes. In the first part of the equation, between 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; and
* 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

**Back propagation Algorithm**

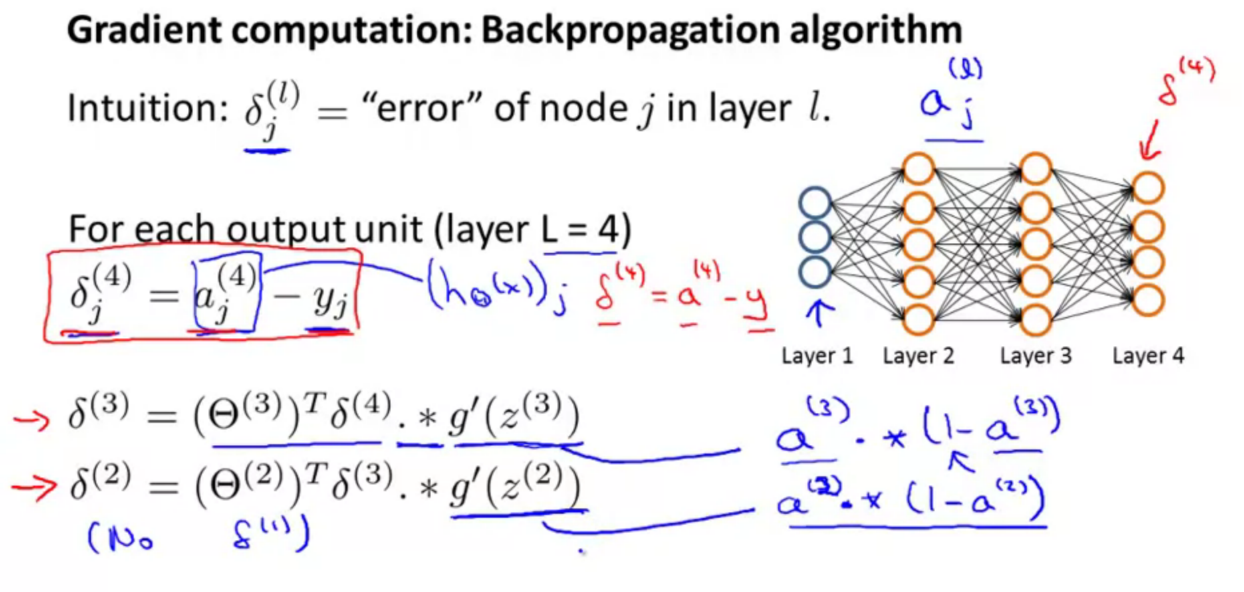
"Backpropagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression.

Our goal is to compute:

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):

In back propagation we're going to compute for every node:



Recall that ​ is activation node j in layer l.

For the **last layer**, we can compute the vector of delta values with:

Where L is our total number of layers and 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:

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:

The full back propagation equation for the inner nodes is then:

A. Ng states that the derivation and proofs are complicated and involved, but you can still implement the above equations to do back propagation without knowing the details.

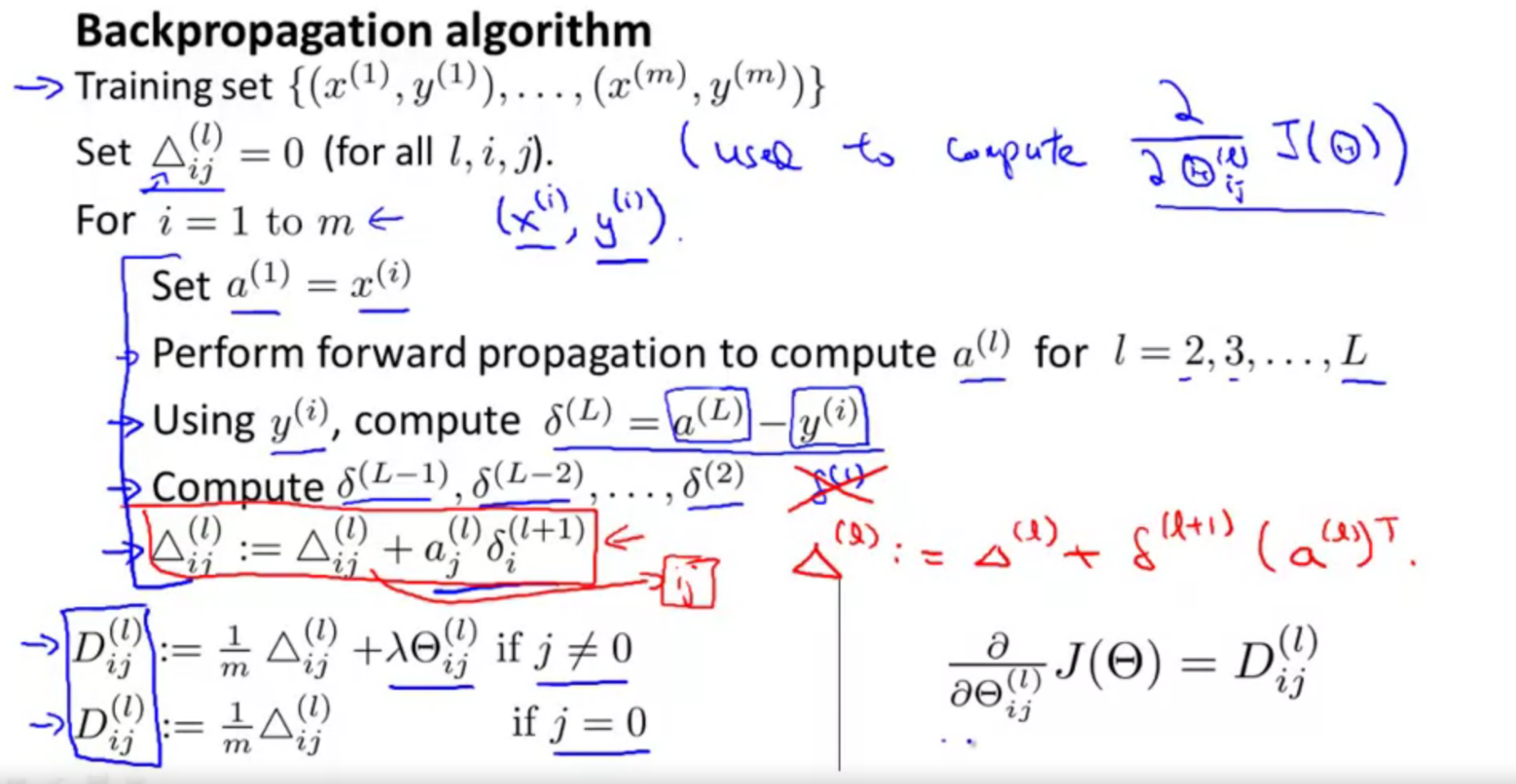
We can compute our partial derivative terms by multiplying our activation values and our error values for each training example t:

This however ignores regularization, which we'll deal with later.

Note: and are vectors with elements. Similarly, is a vector with elements. Multiplying them produces a matrix that is ​ by ​ which is the same dimension as That is, the process produces a gradient term for every element in . (Actually, has column, so the dimensionality is not exactly the same).

We can now take all these equations and put them together into a backpropagation algorithm:

**Back propagation Algorithm**

****

Given training set {

* Set

For training example t =1 to m:

* Set
* A picture containing diagram

  Description automatically generatedPerform forward propagation to compute
* Using , compute
* Compute
* If j≠0 NOTE: Typo in lecture slide omits outside parentheses. This version is correct.
* If j=0

The capital-delta matrix is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative.

The actual proof is quite involved, but, the terms are the partial derivatives and the results we are looking for:

**Backpropagation Intuition**

The cost function is:

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

More intuitively you can think of that equation roughly as:

Intuitively, ​ is the "error" for (unit j in layer l)

More formally, the delta values are actually the derivative of the cost function:

Diagram

Description automatically generatedRecall 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:

In the image above, to calculate ​, we multiply the weights and by their respective values found to the right of each edge. So we get ​ . To calculate every single possible ​, we could start from the right of our diagram. We can think of our edges as our ​ . Going from right to left, to calculate the value of ​, you can just take the over all sum of each weight times the it is coming from. Hence, another example would be ​.

Note: In lecture, sometimes i is used to index a training example. Sometimes it is used to index a unit in a layer. In the Back Propagation Algorithm described here, t is used to index a training example rather than overloading the use of i.

# Implementation Note: Unrolling Parameters

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

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

|  |  |
| --- | --- |
| 2 | deltaVector = [ D1( : ); D2( : ); D3( : ) ] |

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

|  |  |
| --- | --- |
| 1  2  3 | Theta1 = reshape( thetaVector(1:110) , 10, 11)  Theta2 = reshape( thetaVector(111:220) , 10, 11)  Theta3 = reshape( thetaVector(221:231) , 1, 11) |

NOTE: The lecture slides show an example neural network with 3 layers. However, 3 theta matrices are defined: Theta1, Theta2, Theta3. There should be only 2 theta matrices: Theta1 (10 x 11), Theta2 (1 x 11).

To summarize:

Graphical user interface, text, application

Description automatically generated

# Gradient Checking

Gradient checking will assure that our backpropagation works as intended.

We can approximate the derivative of our cost function with:

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

A good small value for (epsilon), guarantees the math above to become true. If the value be much smaller, may we will end up with numerical problems. The professor Andrew usually uses the value

We are only adding or subtracting epsilon to the matrix. In octave we can do it as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | epsilon = 1e-4;  for I in 1:n,  thetaPlus = theta;  thetaPlus(i) += epsilon;  thetaMinus = theta;  thetaMinus(i) -= epsilon;  gradApprox(i) = ( J(thetaPlus) – J(thetaMinus) )/(2\*epsilon)  end; |

We then want to check that gradApprox ≈ deltaVector.

Once you've verified **once** that your backpropagation algorithm is correct, then you don't need to compute gradApprox again. The code to compute gradApprox is 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:

Initialize each ​ to a random value between :

|  |  |
| --- | --- |
| 1  2  3  4 | 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) will initialize a matrix of random real numbers between 0 and 1. (Note: this epsilon is unrelated to the epsilon from Gradient Checking)

Why use this method? This paper may be useful: <https://web.stanford.edu/class/ee373b/nninitialization.pdf>

# 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 total.

* Number of input units = dimension of features
* 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 more than 1 hidden layer, then the same number of units in every hidden layer.

**Training a Neural Network**

1. Randomly initialize the weights
2. Implement forward propagation to get
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 in 1:m,  Perform forward propagation and backward propagation 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:

Chart, surface chart

Description automatically generated

Ideally, you want This will minimize our cost function. However, keep in mind that is not convex and thus we can end up in a local minimum instead.