Week 5 –Training Neural network

Aim:

Earlier programming exercise was done using the theta layer 1 and layer 2 values given to us, here we learn how to train the neural network.

Cost function:

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

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

Recall that in neural networks, we may have many output nodes. We denote $h_\Theta(x)_k$ as being a hypothesis that results in the k^{th} output. Our cost function for neural networks is going to be a generalization of the one we used for logistic regression. Recall that the cost function for regularized logistic regression was:

Logistic regression:

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

Neural network:

$$\rightarrow h_\Theta(x) \in \mathbb{R}^K \quad (h_\Theta(x))_i = i^{th} \text{ output}$$

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1-y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right]$$

$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

K depends on the number of classes, K is binary when only two classes, for multi class classification k is more than 2.

If K = 4, we are summing basically the cost function in logistic regression from k =1 to k= 4

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.

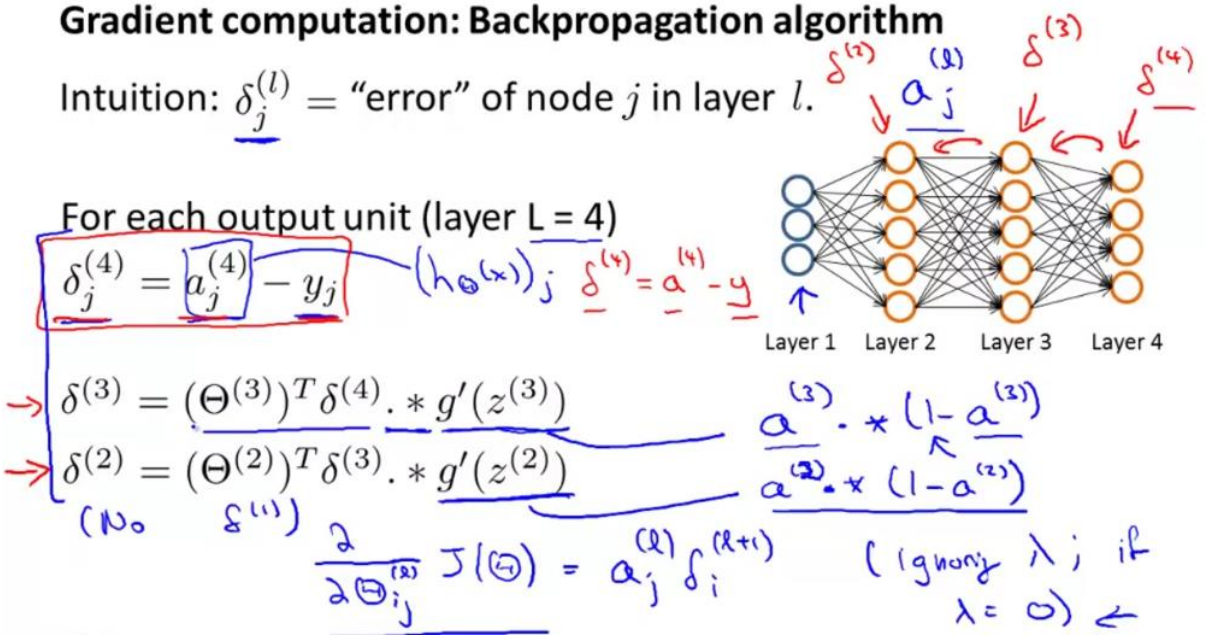
In the regularization part, after the square brackets, we must account for multiple theta matrices. **The number of columns** in our current theta matrix is equal to the **number of nodes**

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:

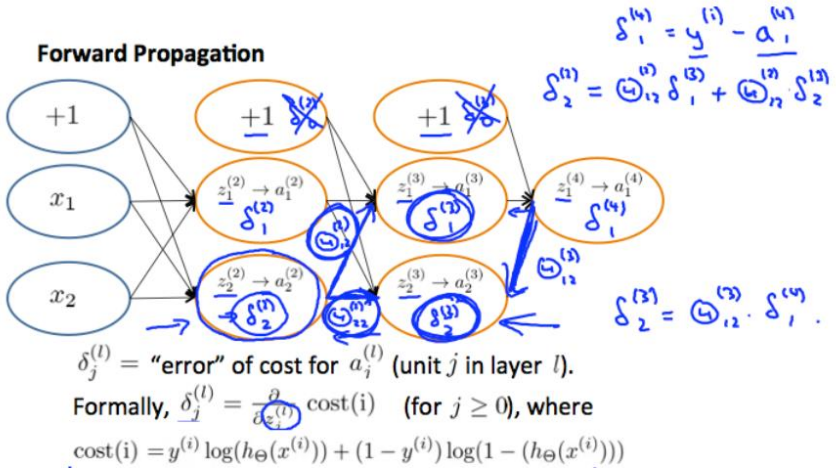
- the double sum simply adds up the logistic regression costs calculated for each cell in the output layer
- The triple sum simply adds up the squares of all the individual Θs in the entire network.
- the i in the triple sum does not refer to training example i

Backpropagation algorithm:



Compute error in each layer, no delta1 as there's no error in layer one, as it's the input unit.

Example calculation



In the image above, to calculate $\delta_2^{(2)}$, we multiply the weights $\Theta_{12}^{(2)}$ and $\Theta_{22}^{(2)}$ by their respective δ values found to the right of each edge. So we get $\delta_2^{(2)}= \Theta_{12}^{(2)}*\delta_1^{(3)}+\Theta_{22}^{(2)}*\delta_2^{(3)}$. To calculate every single possible $\delta_j^{(l)}$, we could start from the right of our diagram. We can think of our edges as our Θ_{ij}. Going from right to left, to calculate the value of $\delta_j^{(l)}$, you can just take the over all sum of each weight times the δ it is coming from. Hence, another example would be $\delta_2^{(3)}=\Theta_{12}^{(3)}*\delta_1^{(4)}$.

Rolling and unrolling vector

- **Theta** = [theta1 (:), theta2 (:), theta3 (:));
- **Reshape to get theta1** = reshape(Theta(1:n), rows, cols); where n = rows*cols

Learning Algorithm

→ Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.
→ Unroll to get `initialTheta` to pass to
→ `fminunc(@costFunction, initialTheta, options)`

`function [jval, gradientVec] = costFunction(thetaVec)`
→ From `thetaVec`, get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ reshape
→ Use forward prop/back prop to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\Theta)$.
Unroll $D^{(1)}, D^{(2)}, D^{(3)}$ to get `gradientVec`.

Numerical gradient checking:

Check if the derivative of cost fcn is right

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

$$\frac{\partial}{\partial\Theta}J(\Theta) \approx \frac{J(\Theta+\epsilon)-J(\Theta-\epsilon)}{2\epsilon}$$

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

$$\frac{\partial}{\partial\Theta_j}J(\Theta) \approx \frac{J(\Theta_1,\ldots,\Theta_j+\epsilon,\ldots,\Theta_n)-J(\Theta_1,\ldots,\Theta_j-\epsilon,\ldots,\Theta_n)}{2\epsilon}$$

A small value for ϵ (epsilon) such as $\epsilon = 10^{-4}$, guarantees that the math works out properly. If the value for 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:

```
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;

```

Implementation Note:

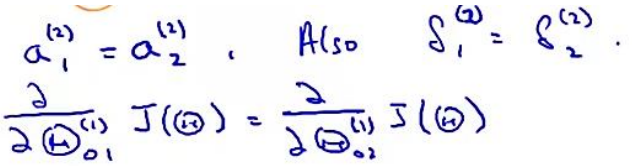
- Implement backprop to compute `DVec` (unrolled $D^{(1)}, D^{(2)}, D^{(3)}$).
- Implement numerical gradient check to compute `gradApprox`.
- Make sure they give similar values.
- Turn off gradient checking. Using backprop code for learning.

Important:

- Be sure to disable your gradient checking code before training your classifier. If you run numerical gradient computation on every iteration of gradient descent (or in the inner loop of `costFunction(...)`) your code will be very slow.

Random initialization

Can't set the theta values to zero, as the hidden activation units will have the same value, a1[2ND layer] = a2 [2ND layer] therefore the delta values will also be the same.



To solve this problem, we use random initialization or symmetry breaking. Set theta1 and theta2 to some random values.

E.g.

```
Theta1 = rand(10,11)*(2*INIT_EPSILON)
         - INIT_EPSILON;

Theta2 = rand(1,11)*(2*INIT_EPSILON)
         - INIT_EPSILON;
```

Training a neural network

1. Randomly initialize weights
2. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
3. Implement code to compute cost function $J(\Theta)$
4. Implement backprop to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$

Perform FP on example #1 ($x^{(1)}$, $y^{(1)}$), then perform BP using the output values, perform FP on example #2 ($x^{(2)}$, $y^{(2)}$), then perform BP and so on.

```
for i = 1:m
```

- Perform forward propagation and backpropagation u example $(x^{(i)}, y^{(i)})$
  (Get activations $a^{(l)}$ and delta terms $\delta^{(l)}$ for $l = 2, \ldots, j$

Putting it together

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

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

Training a Neural Network

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

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

L is the number of layers