[Summary]NNs 2

Cost Function

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

Recall that in nerual networks, we may have many output nodes. We denote $\mathrm{h}_{\Theta}(x)_k$ 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:

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

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

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^i \log\left(\left(\mathrm{h}_\theta\left(x^i\right)\right)_k\right) + \left(1-y_k^i\right)\log\left(1-\left(\mathrm{h}_\theta\left(x^i\right)\right)_k\right)\right]$$
$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\Theta_{j,i}^{\mathrm{l}}\right)^2$$

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.

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:

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

That is, we want to minimize our cost function J using an optimal set of parameters in theta.

In this section we'll look at the equations we use to compute the partial derivative of $J(\Theta)$:

$$\frac{\delta}{\delta \Theta_{i,j}^{l}} J(\Theta)$$

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

$$\delta_j^l = error\ of\ node\ j\ in\ layer\ l$$

Recall that a_j^l is activation node j in layer l.
For the **last layer**, we can compute the vector of delta values with:

$$\delta^L = a^L - y$$

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:

$$\delta^l = ((\Theta^l)^T \delta^{l+1}) .* g'(z^l) \quad note:\ g(z^l) = a^l$$

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:

$$g'(u) = g(u) .* \left(1 - g(u)\right)$$

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

$$\delta^l = ((\Theta^l)^T \delta^{l+1}) .* a^l .* (1 - a^l)$$

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

$$\frac{\delta}{\delta\Theta_{i,j}^{l}}J(\Theta) = \frac{1}{m}\sum_{t=1}^{m} a_j^{(t)(l)}\,\delta_i^{(t)(l+1)}$$

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

Back propagation Algorithm

Given training set $\{(x^1, y^1) \cdots (x^m, y^m)\}$

- $Set\ \Delta_{i,j}^{l} \coloneqq 0\ for\ all\ (l,i,j)$

For training example t = 1 to m:

- $Set\ a^1 \coloneqq x^t$
- $Perform\ forward\ propagation\ to\ compute\ a^l\ for\ l = 2,\ 3,\ \ldots,\ L$
- $Using\ y^t,\ compute\ \delta^L = a^L - y^t$
- $Compute\ \delta^{L-1},\ \delta^{L-2},\ \ldots,\ \delta^2\ using\ \delta^l = \left(\left(\Theta^{l}\right)^T \delta^{l+1}\right) .* a^l .* (1 - a^l)$
- $\Delta_{i,j}^{l} \coloneqq \Delta_{i,j}^{l} + a_j^l \delta_i^{l+1}\ or\ with\ vectorization,\ \Delta^{l} \coloneqq \Delta^{l} + \delta^{l+1}(a^l)^T$
- $D_{i,j}^{l} \coloneqq \frac{1}{m}\left(\Delta_{i,j}^{l} + \lambda\Theta_{i,j}^{l}\right) \quad if\ j \neq 0$
- $D_{i,j}^{l} \coloneqq \frac{1}{m}\Delta_{i,j}^{l} \quad if\ j = 0$

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

$D_{i,j}^{l}$ terms are the partial derivatives and the results we are looking for:

$$D_{i,j}^{l} = \frac{\delta}{\delta \Theta_{i,j}^{l}} J(\Theta)$$

Gradient Checking

Gradient checking will assure that our backpropagation works as intended.

We can approximate the derivative of our cost function with:

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

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

$$\frac{\delta}{\delta \Theta_{\mathrm{j}}} J(\Theta) \approx \frac{J\left(\Theta_1, \ldots, \Theta_{\mathrm{j}} + \epsilon, \ \ldots, \ \Theta_{\mathrm{n}}\right) - \mathrm{J}\left(\Theta_1, \ldots, \Theta_{\mathrm{j}} - \epsilon, \ \ldots, \ \Theta_{\mathrm{n}}\right)}{2\epsilon}$$

A good small value for ϵ (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 $\epsilon = 10^{-4}$.

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 Θ_{ij}^{l} to a random value between $[-\epsilon,\epsilon]$:

$$\epsilon = \frac{\sqrt{6}}{\sqrt{Loutput + Linput}}$$
$$\Theta^{l} = 2\epsilon\, rand(Loutput,\ Linput + 1) - \epsilon$$

rand($\mathrm{Loutput}$,$\mathrm{Linput} + 1$) will initialize a matrix of random real numbers between 0 and 1. (Note: this **epsilon 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 total.

- 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 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 $\mathrm{h}_{\Theta}(x^{i})$
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.