Lecture 55: Gradient Checking

Backpropagation is a relatively complex algorithm for computing $\frac{\partial J}{\partial \Theta_{ij}^{(l)}}$ and there is definitely a chance that our implementation of it may be buggy. Gradient checking is a generic method for checking that the gradients of the cost function J are being computed correctly.

If we have a function of one variable $J(\theta)$, we can estimate its derivate as follows:

$$\frac{dJ}{d\theta}(\theta) \approx \frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon}$$

This is called the two-sided derivative. Sometimes the one-sided derivative is also used: $J'(\theta) \approx \frac{J(\theta+\varepsilon) - J(\theta)}{\varepsilon}$.

Andrew claims the two-sided approximation is more accurate. He also says he usually uses $\varepsilon \approx 10^{-4}$.

Of course, for a function of multiple variables $J(\theta_1, \theta_2, \ldots, \theta_n)$:

$$\frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J(\theta_1+\varepsilon, \theta_2, \ldots, \theta_n) - J(\theta_1-\varepsilon, \theta_2, \ldots, \theta_n)}{2\varepsilon}$$

$$\vdots$$

$$\frac{\partial J}{\partial \theta_n}(\theta) \approx \frac{J(\theta_1, \theta_2, \ldots, \theta_n+\varepsilon) - J(\theta_1, \theta_2, \ldots, \theta_n-\varepsilon)}{2\varepsilon}$$

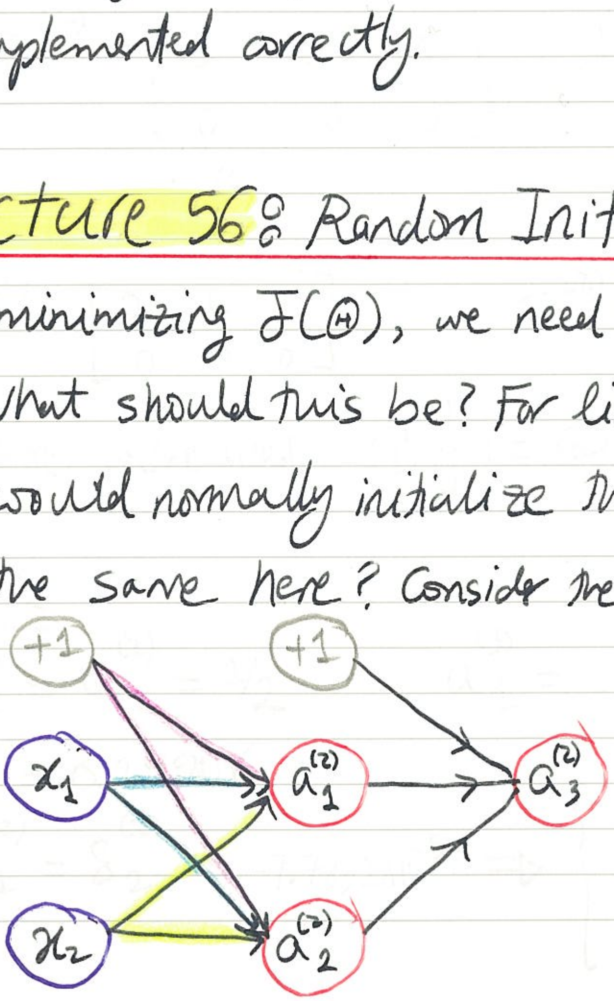
We can use these on the unrolled version of Θ to check that we've implemented backpropagation properly.

Why are we not using the numerical derivat instead of backpropagation to begin with? Because it would be super slow! Backpropagation computes all gradients $\nabla_{\Theta} J$ in one go! If we were to do this using numerical derivatives, we'd need $\underset{\begin{cases}\text{one for } +\varepsilon \\ \text{one for } -\varepsilon\end{cases}}{2} \times \underbrace{\text{\# of parameters}}_{\text{for every partial derivative.}}$ forward propagations.

That's why we only use gradient checking to make sure backpropagation is implemented correctly.

Lecture 56: Random Initialization

For minimizing $J(\Theta)$, we need to supply an initial guess for Θ. What should this be? For linear and logistic regression, we would normally initialize the parameters to zero. Can we do the same here? Consider the following toy example:



Let's step through a few iterations of Gradient Descent, starting with $\Theta^{(l)}_{ij} = 0$. Assume $x_1 = 1/2$, $x_2 = 3/4$, $y = 1/4$, & $\alpha = 0.01$ (α is the learning rate used in Gradient Descent).

Step 1

$$a^{(2)}_1 = a^{(2)}_2 = a^{(3)}_1 = 1/2$$

$$\delta^{(3)}_1 = 1/2 - 1/4 = 1/4$$

$$\delta^{(2)}_1 = \delta^{(2)}_2 = 0$$

$$\Rightarrow \begin{cases} D^{(2)} = \begin{bmatrix} \overset{D^{(2)}_{10}}{1/4} & \overset{D^{(2)}_{11}}{1/8} & \overset{D^{(2)}_{12}}{1/8} \end{bmatrix} \\ D^{(1)} = \begin{bmatrix} 0 & 0 & \overset{D^{(1)}_{12}}{0} \\ \underset{D^{(1)}_{20}}{0} & 0 & 0 \end{bmatrix} \end{cases}$$

$$\Theta^{(1)} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \& \quad \Theta^{(2)} = \begin{bmatrix} -1/400 & -1/800 & -1/800 \end{bmatrix}$$

where we've used $\Theta^{(l)}_{ij} := \Theta^{(l)}_{ij} - \alpha D^{(l)}_{ij}$

Step 2

$$a^{(2)}_1 = a^{(2)}_2 = 1/2 \qquad a^{(3)}_1 = 0.49906$$

$$\delta^{(3)}_1 = 0.24906$$

$$\delta^{(2)}_1 = \delta^{(2)}_2 = -7.7832 \times 10^{-5} \Rightarrow \begin{cases} D^{(2)} = \begin{bmatrix} 0.24906 & 0.12453 & 0.12453 \end{bmatrix} \\ D^{(1)} = \begin{bmatrix} -7.7823 \times 10^{-5} & -3.8916 \times 10^{-5} & -5.8374 \times 10^{-5} \\ -7.7823 \times 10^{-5} & -3.8916 \times 10^{-5} & -5.8374 \times 10^{-5} \end{bmatrix} \end{cases}$$

We see the pattern: the gradients w.r.t all weights coming out of the same input node (those colored the same on page 2) are the same. As a result, we will always have $a_1^{(2)} = a_2^{(2)}$. Now imagine we had many more units in layer 2. We would still have $a_1^{(2)} = a_2^{(2)} = \cdots = a_{s_2}^{(2)}$. This is a highly redundant representation, and is caused by our initial guess for Θ.

For this purpose, we initialize $\Theta_{ij}^{(l)}$ by picking random numbers in some range $[-\varepsilon, +\varepsilon]$. This is also called "symmetry breaking", because it breaks the symmetry of having the same weights.

Note that there's nothing special about zero; the problem of the symmetric weights would arise if all eweights are initialized to the same number. In fact, in the example we've been looking at, if we initialize $\Theta^{(1)}$ to one number & $\Theta^{(2)}$ to another, we would still not break the symmetry.

In the programming exercise, $\varepsilon = 0.12$ is used.

Lecture 57: Putting It Together

The first decision to make when training a neural network is to pick a network architecture: how many hidden layers? How many neurons per hidden layer? Here are some general guidelines:

* A reasonable default for the number of hidden layers is 1.
* If there is more than 1 hidden layer, they should all have the same number of units (again, as a default choice).
* Usually the more hidden units the better. Typically, the # of hidden units is on the same order as the # of input units, perhaps a few times more.

In later lectures we'll say a lot more about network architecture.

Let's summarize the steps needed to train 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 $\nabla_\Theta J$.
5. Use gradient checking to make sure backprop is implemented correctly.
6. Use gradient descent or other optimization method to minimize $J(\Theta)$.

It's important to note that $J(\Theta)$ is non-convex, so there's no guarantee that our optimization algorithm, like gradient descent, would find the global minima. Andrew claims that this does not turn out to be a problem in practice. Even if a local minimum is found, it's usually good enough.

Lecture 58: Autonomous Driving

This lecture shows a super cool (and relatively old) video of a neural network which learns to drive a car. The input data is an image of the road, and the output is the steering direction. First it's trained by a human driver & then given autonomy to drive on its own. A neural network with only one layer is used!