Debugging: Gradient Checking

So far we have worked with relatively simple algorithms where it is straight-forward to compute the objective function and its gradient with pen-and-paper, and then implement the necessary computations in MATLAB. For more complex models that we will see later (like the back-propagation method for neural networks), the gradient computation can be notoriously difficult to debug and get right. Sometimes a subtly buggy implementation will manage to learn something that can look surprisingly reasonable (while performing less well than a correct implementation). Thus, even with a buggy implementation, it may not at all be apparent that anything is amiss. In this section, we describe a method for numerically checking the derivatives computed by your code to make sure that your implementation is correct. Carrying out the derivative checking procedure described here will significantly increase your confidence in the correctness of your code.

Suppose we want to minimize $J(\theta)$ as a function of θ. For this example, suppose $J : \Re \mapsto \Re$, so that $\theta \in \Re$. If we are using `minFunc` or some other optimization algorithm, then we usually have implemented some function $g(\theta)$ that purportedly computes $\frac{d}{d\theta}J(\theta)$.

How can we check if our implementation of g is correct?

Recall the mathematical definition of the derivative as:

$$\frac{d}{d\theta}J(\theta) = \lim_{\epsilon \to 0} \frac{J(\theta+\epsilon) - J(\theta-\epsilon)}{2\epsilon}.$$

Thus, at any specific value of θ, we can numerically approximate the derivative as follows:

$$\frac{J(\theta+\mathrm{EPSILON}) - J(\theta-\mathrm{EPSILON})}{2 \times \mathrm{EPSILON}}$$

In practice, we set $\mathrm{EPSILON}$ to a small constant, say around 10^{-4}. (There's a large range of values of $\mathrm{EPSILON}$ that should work well, but we don't set $\mathrm{EPSILON}$ to be "extremely" small, say 10^{-20}, as that would lead to numerical roundoff errors.)

Thus, given a function $g(\theta)$ that is supposedly computing $\frac{d}{d\theta}J(\theta)$, we can now numerically verify its correctness by checking that

$$g(\theta) \approx \frac{J(\theta+\mathrm{EPSILON}) - J(\theta-\mathrm{EPSILON})}{2 \times \mathrm{EPSILON}}.$$

The degree to which these two values should approximate each other will depend on the details of J. But assuming $\mathrm{EPSILON} = 10^{-4}$, you'll usually find that the left- and right-hand sides of the above will agree to at least 4 significant digits (and often many more).

Now, consider the case where $\theta \in \Re^n$ is a vector rather than a single real number (so that we have n parameters that we want to learn), and $J : \Re^n \mapsto \Re$. We now generalize our derivative checking procedure to the case where θ may be a vector (as in our linear regression and logistic regression examples). If ever we are optimizing over several variables or over matrices, we can always pack these parameters into a long vector and use the same method here to check our derivatives. (This will often need to be done anyway if you want to use off-the-shelf optimization packages.)

Suppose we have a function $g_i(\theta)$ that purportedly computes $\frac{\partial}{\partial \theta_i}J(\theta)$; we'd like to check if g_i is outputting correct derivative values. Let $\theta^{(i+)} = \theta + \mathrm{EPSILON} \times \vec{e}_i$, where

$$\vec{e}_i = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

is the i-th basis vector (a vector of the same dimension as θ, with a "1" in the i-th position and "0"s everywhere else). So, $\theta^{(i+)}$ is the same as θ, except its i-th element has been incremented by $\mathrm{EPSILON}$. Similarly, let $\theta^{(i-)} = \theta - \mathrm{EPSILON} \times \vec{e}_i$ be the corresponding vector with the i-th element decreased by $\mathrm{EPSILON}$.

We can now numerically verify $g_i(\theta)$'s correctness by checking, for each i, that:

$$g_i(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2 \times \mathrm{EPSILON}}.$$

Gradient checker code

As an exercise, try implementing the above method to check the gradient of your linear regression and logistic regression functions. Alternatively, you can use the provided `ex1/grad_check.m` file (which takes arguments similar to `minFunc`) and will check $\frac{\partial J(\theta)}{\partial \theta_i}$ for many random choices of i.

Self-Taught Learning

Self-Taught Learning (http://ufldl.stanford.edu/tutorial/selftaughtlearning/SelfTaughtLearning)

Exercise: Self-Taught Learning (http://ufldl.stanford.edu/tutorial/selftaughtlearning/ExerciseSelfTaughtLearning)