The Hessian to Widrow-Hoff: Calculus and Machine Learning

1 Introduction

Optimization problems are an integral part of both calculus and computer science, and they can range from the extremely trivial to the unimaginably complicated. Here I'll focus on somewhere in-between; on a specific set of problems that have practical applications in machine learning and some other issues we might face. I'm going to use a simple machine learning algorithm as an example to walk through this problem, which I've adapted from Stanford's excellent CS229 course. It turns out it's a great fit for what I want to talk about and try to implement. There'll be a bit of an introduction to linear regression, and then we'll slowly work our way toward the calculus, with a decent conceptual basis to work from.

2 LMS Linear Regression

So here's the task: we want to make a guess on the price of a given house in Manhattan. We, of course, know some things about that house, such as the number of bedrooms, the number of bathrooms, and the square footage. What we don't know is the actual selling price of the house. So we need to form some model in order to best approximate this value, maybe based on the selling price of houses in the area. These are called training examples – values we, as the programmer or user, input into the model. The model should then adjust its parameters accordingly, and, when given the properties of our house, spit out a value ("hypothesis") which corresponds to the expected selling price. The simplest model to use is linear regression; with which the expression for the hypothesis would be as follows:

$$h = \theta_0 + \sum_{i=1}^{n} x_i \theta_i \tag{1}$$

where the θ_i are the parameters of our n-parameter model, and the x_i are the inputs. h is then a function of our x_i. Noting that we let $x_0 = 1$ so that we can get a 'DC' value θ_0, we can express this as:

$$h(\begin{pmatrix} x_1 & x_2 & \dots & x_n \end{pmatrix}) = \begin{pmatrix} x_0 & x_1 & \dots & x_n \end{pmatrix} \begin{pmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \dots \\ \theta_n \end{pmatrix} \tag{2}$$

Or, more concisely, using vectors $\mathbf{x}$ and θ:

$$h(\mathbf{x}) = \mathbf{x}\theta^T \tag{3}$$

So assuming we have some good values for θ, we should be able to form a reasonably good approximation of our house price. The problem, of course, is finding these θ_i values!

Our training examples, to find these θ_i, are denoted $(\mathbf{x}^j, y^j)$. The js do not denote exponentiation, but rather are indices of the training examples (i.e. the first training example is $(\mathbf{x}^0, y^0)$, the second $(\mathbf{x}^1, y^1)$, etc. For these training examples, our goal is to minimize the difference between our hypotheses $h(\mathbf{x}^j)$ and the y^j we are given. We can define an error function, $J(\theta)$, for one training example as follows, assuming we're using least mean squares (LMS) regression:

$$J_j(\theta) = \frac{1}{2}(y^j - h(\mathbf{x}^j))^2 = \frac{1}{2}(y^j - \mathbf{x}^j\theta^T)^2 \tag{4}$$

... and for m training examples, just the sum of the individual errors:

$$J(\theta) = \frac{1}{2}\sum_{j=1}^{m}(y^j - h(\mathbf{x}^j))^2 \tag{5}$$

We'll focus on just the first one for now; with just one training example. Our goal here is to minimize the difference between the actual value and our hypothesis, by modifying the values of θ. Essentially, we want to minimize $J_j(\theta)$. Now here comes a bit of calculus - remember how to optimize a function? Yeah, pretty simple, just take the derivative and set it to zero. Sounds simple enough, but of course, we have multiple θ_i, and so we need to minimize the error for each θ_i.

3 The Calculus

Now that the context is set, it's time for the calculus interlude. I'll just do a brief overview, but I'm assuming you all know how to find the local maxima and minima of a function $f : \mathbb{R} \to \mathbb{R}$. If x is your dependent variable, just take $\frac{\mathrm{d}f}{\mathrm{d}x} = 0$ and solve for x. To determine whether these values correspond to maxima or minima, we take $\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}$ and find how the concavity changes at the point we're interested in.

The situation gets a bit more complicated when we have a function $f : \mathbb{R}^n \to \mathbb{R}$. Firstly, we can't just take a single derivative; we have multiple partial derivatives -; multiple variables we need to optimize for. So instead of a single expression, we get a vector $\langle \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \rangle$. This is known as the ¡strong¿gradient¡/strong¿, and it indicates the direction of greatest increase of the function. A more concise notation can be acheived using the del (∇) operator, where $\nabla = \langle \frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \ldots, \frac{\partial}{\partial x_n} \rangle$, and so we can express the gradient of a function f as ∇f.

Setting the gradient to zero and solving for your variables will give you the points at which the change in f is zero; i.e. at the local maxima or minima. Actually, that isn't quite true. You don't necessarily have a maximum or minimum where the gradient is zero; you can have saddle points where the function doesn't lsquo;change' but is also not at its highest or lowest in its neighbourhood.

So how do we know if a critical point (a point where $\nabla f = 0$) is a local maximum, minimum, or saddle point? Here it gets a bit more complicated. Just as we don't have one singular derivative, we don't have one second derivative. In fact, for n variables in f, we have n^2 second derivatives! This becomes apparent once you realize what we're doing is taking the gradient of each element in the gradient vector. We put these values into what is called a Hessian matrix, as follows:

$$Hf = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \tag{6}$$

The determinant of this function, $|\det(Hf)|$, can give us information on the classification of the critical point. We'll focus on functions $f : \mathbb{R}^2 \to \mathbb{R}$ in this discussion. Our determinant is then $\frac{\partial^2 f}{\partial x^2}\frac{\partial^2 f}{\partial y^2} - (\frac{\partial^2 f}{\partial x \partial y})^2$, and we can state the following conditions to classify the point:

1. If $|\det(Hf)| > 0$ and $\frac{\partial^2 f}{\partial x^2} > 0$, the point is a local minimum
2. Else if $|\det(Hf)| > 0$ and $\frac{\partial^2 f}{\partial x^2} < 0$, the point is a local minimum
3. Else if $|\det(Hf)| < 0$, the point is a saddle point
4. Else, this test is inconclusive.

4 Deriving Widrow-Hoff

So now we know how to find the local maxima and minima of a multivariable function! Returning to our exploration of linear regression, how exactly does this help us? The answer is... not much. We're dealing with a numerical, not analytical, question, and it doesn't necessarily help to have a deep knowledge of what a Hessian matrix is, or how to classify a critical point. In LMS linear regression, we conveniently only have one critical point; a local minimum. We just need to find this point.

If we 'follow' the gradient until we reach a point where the gradient is zero, we are, to abuse terms, 'ascending' the function until we reach a peak. If we follow the negative gradient, we should reach a local minimum. The algorithm which accomplishes this feat is called the *gradient descent* algorithm, and it is derived as follows:

Recall:

$$J(\theta) = \frac{1}{2}(y - h(\mathbf{x}))^2 \tag{7}$$

$$\begin{aligned}
\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{1}{2}\frac{\partial}{\partial \theta_j}(y - h(\mathbf{x}))^2 \\
&= \frac{1}{2} \cdot 2 \cdot (y - h(\mathbf{x}))\frac{\partial}{\partial \theta_j}(y - h(\mathbf{x})) \\
&= (y - h(\mathbf{x}))\frac{\partial}{\partial \theta_j}\sum_{i=0}^{n}(y - \theta_i x_i) \\
&= (y - h(\mathbf{x}))x_j
\end{aligned} \tag{8}$$

It seems intuitive that to get a value for θ that is closer to the local minimum, we should adjust our values such that it "follows" the negative of the gradient; i.e.

$$\theta_j := \theta_j - \nabla_{\theta_j} J(\theta) \tag{9}$$

And so we get the **Widrow-Hoff learning rule**

$$\theta_j := \theta_j - (y - h(\mathbf{x}))x_j \tag{10}$$

We adjust our θ value by adding to it the negative of the gradient we derived above, which brings it closer and closer to the local minimum -; until the difference between the previous and new values of θ_j tends to zero (when the term converges). We can adjust the rate of descent by adding a parameter α;

$$\theta_j := \theta_j - \alpha(y - h(\mathbf{x}))x_j \tag{11}$$

Which is known as the learning rate. For multiple training examples, we simply modify our update rule to be

$$\theta_j := \theta_j - \alpha\sum_{i=1}^{m}(y^i - h(\mathbf{x}^i))x_j^i \tag{12}$$

And repeat until convergence. This is known as **batch gradient descent**.

So we've gone from a practical problem, to an interlude of calculus, to a practical solution. And this is just the start of the many machine learning techniques; you can use locally weighted regression (LOWESS), or local regression with k-means clustering, or move toward unsupervised learning algorithms. It's an amazing field, and Andrew Ng's CS229 course (from which I've adapted the machine learning part of this post - not the calculus part) is a great start.