Hessian Free Optimization

Thursday, February 13, 2014

Training a neural network involves minimizing its error with respect to its parameters, and since larger neural networks may have millions of parameters, this poses quite a challenge. Surprisingly, many of the recent advances in neural networks come not from improved data processing, neural network architectures, or other machine learning tricks, but instead from incredibly powerful methods for function minimization.

In this notebook, we'll go through several optimization methods and work our way up to Hessian Free Optimization (https://machinelearning.wustl.edu/mlpapers/paper_files/icml2010_Martens10.pdf), a powerful optimization method adapted to neural networks by James Martens in 2010.

Gradient Descent

The simplest algorithm for iterative minimization of differentiable functions is known as just **gradient descent**. Recall that the gradient of a function is defined as the vector of partial derivatives:

$$\nabla f(x) = \langle \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \rangle$$

and that the gradient of a function always points towards the direction of maximal increase at that point.

Equivalently, it points *away* from the direction of maximum decrease - thus, if we start at any point, and keep moving in the direction of the negative gradient, we will eventually reach a local minimum.

This simple insight leads to the Gradient Descent algorithm. Outlined algorithmically, it looks like this:

> 1. **Initialization**: Pick a point x_0 as your initial guess.
> 2. **Gradient**: Compute the gradient at your current guess: $v_i = \nabla f(x_i)$
> 3. **Update**: Move by α (your step size) in the direction of that gradient: $x_{i+1} = x_i - \alpha v_i$
> 4. **Iterate**: Repeat steps 1-3 until your function is close enough to zero (until $f(x_i) < \varepsilon$ for some small tolerance ε)

Note that the step size, α, is simply a parameter of the algorithm and has to be fixed in advance. Note also that this is a *first-order* method; that is, we only look at the first derivatives of our function. This will prove to be a significantly limitation of this method, because it means we are assuming that the error surface of the neural network locally looks and behaves like a plane. We are ignoring any and all curvature of the surface, which, as we will see later, may lead us astray and cause our training to progress very slowly.

Newton's Method

Although gradient descent works (in theory), it turns out that in practice, it can be rather slow. To rectify this, we can use information from the *second* derivative of a function. The most basic method for second order minimization is Newton's method. We will first work through Newton's method in one dimension, and then make a generalization for many dimensions.

One Dimension

Suppose we have some function, $f : \mathbb{R} \to \mathbb{R}$. In order to find it's minimum, we want to find the zero of it's derivative, because we know that $f'(x) = 0$ at the minimum of $f(x)$. Let's approximate f with a second-order Taylor expansion about some point x_0:

$$f(x_0 + x) \approx f(x_0) + f'(x_0)x + f''(x_0)\frac{x^2}{2}.$$

We'd like to choose x so that $f(x_0 + x)$ is a minimum. Thus, we do what we usually do - we take the derivative of this expansion with respect to x and set it to zero. We find that

$$\frac{d}{dx}\left(f(x_0) + f'(x_0)x + f''(x_0)\frac{x^2}{2}\right) = f'(x_0) + f''(x_0)x = 0 \implies x = -\frac{f'(x_0)}{f''(x_0)}.$$

If f is just a quadratic function, this would be the absolute minimum. However, we're trying to iteratively find the minimum of *any* nonlinear f, so this may not be the minimum. In order to get to the minimum, we just repeat this process. We start at some guess x_0, and then get closer and closer with the update rule

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)} = x_n - (f''(x_n))^{-1} f'(x_n).$$

This eventually converges to a minimum.

Multiple Dimensions

The algorithm above works only for a single dimension. However, suppose that our function is $f : \mathbb{R}^n \to R$. In that case, we can do the exact same derivation, but replacing derivatives with gradients and second derivatives with Hessians (the matrix of second derivatives):

$$\begin{gathered} f'(x) \to \nabla f(x) \\ f''(x) \to H(f)(x) \end{gathered}$$

Thus, our update rule becomes

$$x_{n+1} = x_n - (H(f)(x_n))^{-1} \nabla f(x_n).$$

which is the optimization algorithm commonly cited as Newton's method.

Problems with Newton's Method

As we can see, Newton's method is a *second* order algorithm, and may perform better than the simpler gradient descent. In a single step, instead of just advancing towards the minimum, it steps towards the global minimum under the assumption that the function is actually quadratic and its second order expansion is a good approximation. This allows it to make big steps in low-curvature scenarios (where $f''(x_n)$ is small) and small steps in high-curvature scenarios. As with gradient descent, we see that the direction we move in is against the gradient ($-\nabla f(x_n)$).

However, Newton's method has a very large drawback: it requires computation of the Hessian matrix H. The issues with this are twofold. First of all, we may not *know* how to compute the Hessian matrix! In a neural network, backpropagation can be used to compute the gradient, but computing the Hessian requires a different algorithm entirely - if it is even possible without resorting to methods of finite differences. The second issue is that if we have n dimensions, the Hessian requires $O(n^2)$ storage space and computation to find, since it is an $n \times n$ matrix.

As we will see, the method of Hessian-free optimization addresses the both of these issues quite neatly, coming up with a general solution to the second issue and a neural-network specific solution to the first one.

The main idea behind Hessian-free optimization is that we can use the insights from Newton's method but come up with a better way to minimize the quadratic function we get. We start off just like Newton's method. Given our function f, we approximate it with a second-order Taylor expansion:

$$f(x + \Delta x) \approx f(x) + \nabla f(x)^T \Delta x + \Delta x^T H(f) \Delta x.$$

We then find the minimum of this approximation (find the best Δx), move to $x + \Delta x$, and iterate until we have convergence.

One of the key ideas, thus, is the way in which we find the minimum of this quadratic function. In order to minimize it, we'll use an iterative method known as *conjugate gradient*.

Conjugate Gradient

Although conjugate gradient is a method minimizing a quadratic function, note that there exist variants for quickly solving systems of differential equations as well as minimizing general nonlinear functions. However, we only need the quadratic minimization algorithm (even though our ultimate goal is general non-linear optimization).

Suppose we have some quadratic function

$$f(x) = \frac{1}{2}x^T Ax + b^T x + c$$

for $x \in \mathbb{R}^n$ with $A \in \mathbb{R}^{n \times n}$ and $b, c \in \mathbb{R}^n$.

We can write any quadratic function in this form, as this generates all the coefficients $x_i x_j$ as well as linear and constant terms. In addition, we can assume that $A = A^T$ (A is symmetric). (If it were not, we could just rewrite this with a symmetric A, since we could take the term for $x_i x_j$ and the term for $x_j x_i$, sum them, and then have $A_{ij} = A_{ji}$ both be half of this sum.)

Taking the gradient of f, we obtain

$$\nabla f(x) = Ax + b,$$

which you can verify by writing out the terms in summation notation.

If we evaluate $-\nabla f$ at any given location, it will give us a vector pointing towards the direction of steepest descent. This gives us a natural way to start our algorithm - pick some initial guess x_0, compute the gradient $-\nabla f(x_0)$, and move in that direction by some step size α. Unlike normal gradient descent, however, we do not have a fixed step size α - instead, we perform a line search in order to find the *best* α. This α is the value of α which brings us to the minimum of f if we are constrainted to move in the direction given by $d_0 = -\nabla f(x_0)$.

Note that computing α is equivalent to minimizing the function

$$\begin{aligned}
g(\alpha) &= f(x_0 + \alpha d_0) \\
&= \frac{1}{2}(x_0 + \alpha d_0)^T A(x_0 + \alpha d_0) + b^T(x_0 + \alpha d_0) + c \\
&= \frac{1}{2}\alpha^2 {d_0}^T A d_0 + {d_0}^T(Ax_0 + b)\alpha + (\frac{1}{2}{x_0}^T A x_0 + {x_0}^T d_0 + c)
\end{aligned}$$

Since this is a quadratic function in α, it has a unique global minimum or maximum. Since we assume we are not at the minimum and not at a saddle point of f, we assume that it has a minimum.

The minimum of this function occurs when $g'(\alpha) = 0$, that is, when

$$g'(\alpha) = ({d_i}^T A d_i)\alpha + {d_i}^T (A x_i + b) = 0.$$

Solving this for α, we find that the minimum is at

$$\alpha = -\frac{{d_i}^T (A x_i + b)}{{d_i}^T A d_i}.$$

Note that since the directon is the negative of the gradient, a.k.a. the direction of steepest descent, α will be non-negative. These first steps give us our second point in our iterative algorithm:

$$x_1 = x_0 - \alpha \nabla f(x_0)$$

If this were simple gradient descent, we would iterate this procedure, computing the gradient at each next point and moving in that direction. However, this has a problem - by moving α_0 in direction d_0 (to find the minimum in direction d_0) and then moving α_1 in direction d_1, we may *ruin* our work from the previous iteration, so that we are no longer at a minimum in direction d_0. In order to rectify this, we require that our directions be *conjugate* to one another.

We define two vectors x and y to be conjugate with respect to some semi-definite matrix A if $x^T A y = 0$. (Semi-definite matrices are ones where $x^T A x \geq 0$ for all x, and are what we require for conjugate gradient.)

Since we have already moved in the $d_0 = -\nabla f(x_0)$ direction, we must find a new direction d_1 to move in that is conjugate to d_0. How do we do this? Well, let's compute d_1 by starting with the gradient at x_1 and then subtracting off anything that would counter-act the previous direction:

$$d_1 = -\nabla f(x_1) + \beta_0 d_0.$$

This leaves us with the obvious question - what is β_0? We can derive that from our definition of conjugacy. Since d_0 and d_1 must be conjugate, we know that ${d_1}^T A d_0 = 0$. Expanding d_1 by using its definition, we get that ${d_1}^T A d_0 = -\nabla f(x_1)^T A d_0 + \beta_0 {d_0}^T A d_0 = 0$. Therefore, we must choose β_0 such that

$$\beta_0 = \frac{\nabla f(x_1)^T A d_0}{{d_0}^T A d_0}.$$

Choosing this β gives us a direction conjugate to all previous directions. Interestingly enough, iterating this will *keep* giving us conjugate directions. After generating each direction, we find the best α for that direction and update the current estimate of position.

Thus, the full Conjugate Gradient algorithm for quadratic functions:

> Let f be a quadratic function $f(x) = \frac{1}{2}x^T Ax + b^T x + c$ which we wish to minimize.
>
> 1. **Initialize:** Let $i = 0$ and $x_i = x_0$ be our initial guess, and compute $d_i = d_0 = -\nabla f(x_0)$.
> 2. **Find best step size:** Compute α to minimize the function $f(x_i + \alpha d_i)$ via the equation
>
> $$\alpha = -\frac{{d_i}^T(Ax_i + b)}{{d_i}^T Ad_i}.$$
>
> 3. **Update the current guess:** Let $x_{i+1} = x_i + \alpha d_i$.
> 4. **Update the direction:** Let $d_{i+1} = -\nabla f(x_{i+1}) + \beta_i d_i$ where β_i is given by
>
> $$\beta_i = \frac{\nabla f(x_{i+1})^T Ad_i}{{d_i}^T Ad_i}.$$
>
> 5. **Iterate:** Repeat steps 2-4 until we have looked in n directions, where n is the size of your vector space (the dimension of x).

Hessian-Free Optimization

We now have all the prerequisite background to understand the Hessian-free optimization method. The general idea behind the algorithm is as follows:

> Let f be any function $f : \mathbb{R}^n \to \mathbb{R}$ which we wish to minimize.
>
> 1. **Initialize:** Let $i = 0$ and $x_i = x_0$ be some initial guess.
> 2. **Set up quadratic expansion:** At the current x_n, compute the gradient $\nabla f(x_n)$ and Hessian $H(f)(x_n)$, and consider the following Taylor expansion of f:
>
> $$f(x + \Delta x) \approx f(x) + \nabla f(x)^T \Delta x + \Delta x^T H(f) \Delta x.$$
>
> 3. **Conjugate gradient:** Compute x_{n+1} using the Conjugate Gradient algorithm for quadratic functions on the current Taylor expansion.
> 4. **Iterate:** Repeat steps 2 and 3 until x_n has converged.

This algorithm summarizes all the necessary ideas, but fails to pass a closer inspection. Let's look at each of the issues with this algorithm and see if we can fix this.

Initialization

Question: In step (1), we need to choose some initial x_0. What do we choose?

Answer: This question comes up with any iterative minimization algorithm. At the very least, there is a simple solution - choose randomly. Sample your parameter values from a normal distribution with zero mean and a relatively small standard deviation. In the case of neural networks, there are better choices, but even then a random initialization will work.

Computing the Hessian

Question: Looking at step (2) of the algorithm, it seems that in order to compute the second order expansion

$$f(x+\Delta x) \approx f(x) + \nabla f(x)^T \Delta x + \Delta x^T H(f) \Delta x.$$

we need to be able to compute the Hessian matrix $H(f)$. However, in the section on Newton's method, we discussed that this was a *major* issue and rendered Newton's method unusable. What's going on? How do we address this?

Answer: In Newton's method, we needed the Hessian matrix H because we had the update rule

$$x_{n+1} = x_n - H^{-1}(x_n) \nabla f(x_n).$$

In each step, we had to compute the Hessian and invert it in order to get the next value. However, for the purposes of the conjugate gradient we're using in step (2), we *don't actually need* the Hessian. All we need is the ability to *use* the Hessian H to compute Hv for some vector v. This is a crucial insight, as it turns out that computing Hv is *way* easier than computing H!

To see this, consider the first component of Hv. The ith row of the Hessian contains partial derivatives of the form $\frac{\partial^2}{\partial x_i x_j} f$, where j is the column index. As per normal matrix-vector multiplication, the ith row of Hv is the dot product of v and ith row of H. The ith row of H can be expressed as the gradient of the derivative of f with respect to x_i, yielding:

$$(Hv)_i = \sum_{j=1}^{N} \frac{\partial^2 f}{\partial x_i x_j}(x) \cdot v_j = \nabla \frac{\partial f}{\partial x_i}(x) \cdot v.$$

This is just the *directional derivative* of $\frac{\partial f}{\partial x_i}$ in the direction v. Recall that the directional derivative is defined as

$$\nabla_v f = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon v) - f(x)}{\varepsilon}$$

Using finite differences, we can approximate this as

$$\nabla_v f \approx \frac{f(x+\varepsilon v) - f(x)}{\varepsilon},$$

for some small ε.

Therefore, we can approximate $(Hv)_i$ using finite differences with $\frac{\partial f}{\partial x_i}$. When we bundle this all up into vector form, we find that

$$Hv \approx \frac{\nabla f(x+\varepsilon v) - \nabla f(x)}{\varepsilon}.$$

Thus, we can compute the Hessian times any vector with just two computations of the gradient, and never need to compute the Hessian itself. Also, for the specific case of neural networks, we can compute Hv directly (https://bcl.hamilton.ie/~barak/papers/nc-hessian.pdf) with an algorithm similar to normal backpropagation.

Applying Conjugate Gradient

Question: Conjugate gradient completes after we do n iterations of the algorithm. However, for neural networks, n may be prohibitively large - potentially, in the millions. What do we do?

Answer: It turns out that in practice, conjugate gradient makes very large improvements in the first few iterations. Thus, we can either terminate it after some fixed number of iterations (a few hundred, perhaps), or use a stopping heuristic to determine approximate convergence.

Conclusion

With the modifications from the previous section, the Hessian Free method becomes applicable to many-dimensional optimization problems. In his paper (https://machinelearning.wustl.edu/mlpapers/paper_files/icml2010_Martens10.pdf), James Martens discusses a number of other optimizations and modifications to that method, but they rely on the exact same structure and intuition as the unmodified method presented above.

Thursday, February 13, 2014 - Posted in machine-learning (https://andrew.gibiansky.com/blog/categories/machine-learning), mathematics (https://andrew.gibiansky.com/blog/categories/mathematics)

« Conjugate Gradient (https://andrew.gibiansky.com/blog/machine-learning/conjugate-gradient)
Fully Connected Neural Network Algorithms » (https://andrew.gibiansky.com/blog/machine-learning/fully-connected-neural-networks)

Contact

If you've got questions, comments, suggestions, or just want to talk, feel free to email me at andrew.gibiansky on Gmail.