J. R. "Bob" Dobbs Memorial Inverse Hessian-Vector Multiplication Method

Howon Lee

January 7 2019

I'm not saying that machine learning is the portal to a demon universe, I'm just saying that some doors are best left unopened. - James Mickens

1 Abstract

You care about multiplication of the inverse Hessian with a vector because Newton's method for optimization demands it. In many applications, the straightforward inverse Hessian multiplication is not possible because of time and space constraints. A method linear to the size of the gradient function in both time and space is given which is very close to the Hessian-vector multiplication of [2], with differential operator and all. However, I have failed to get it working on neural nets for a reason also given.

2 Introduction

The Hessian is the matrix of second derivatives of a function. In Newton's method for optimizations, it is the second derivative with respect to an optimization target. But Newton's method requires the multiplicative inverse of the Hessian (in one dimension, it requires the multiplicative inverse of the second derivative). This led to Newton's method being computationally infeasible for most optimizational regimes where the dimensionality is high, because if the dimension of the system is n (meaning the cardinality of the Hessian is $O(n^2)$), the naive inversion of the Newton's method takes $O(n^2)$ space and $O(n^3)$ time.

There is a shortcut which already exists which is usually more used in neural networks. In neural networks it was introduced by [2]. It consists in noting that the Hessian is a Jacobian itself and finding the Hessian in the Taylor approximation of the gradient about a point. This means that the Hessian-vector multiplication can be found like a sparse matrix-vector multiplication, linear to the order of the vector. However this gives a Hessian-vector multiplication only, not an inverse Hessian-vector multiplication, so one must use Krylov methods to get optimization done[4]. In the following, an inverse Hessian-vector multiplication is shown.

3 J. R. "Bob" Dobbs Memorial Method

Recall the Hessian is a Jacobian, and it is found in the Taylor expansion of the gradient of a function about a point. The crux of the method, almost trivial as it is, is to realize that the inverse Hessian is itself a Jacobian and it is found in the Taylor expansion of the functional inverse of the gradient of a function about a point.

3.1 Inverse Hessian is a Jacobian

The inverse function theorem [1] gives a formula for the matrix inverse of a Jacobian, with conditions (Jacobian determinant must be nonzero). That is, the Jacobian of the (functional) inverse of a function is the multiplicative inverse of the Jacobian of the original function. There is no guarantee for the whole domain.

The Hessian is the Jacobian of the gradient:

$$H(f(x)) = J(\nabla f(x))^T$$

So, the multiplicative inverse of the Hessian is the Jacobian of the functional inverse of the gradient:

$$H^{-1}(f(x)) = J(\nabla^{-1} f(x))^T$$

3.2 Inverse Hessian is found in the Taylor Expansion of ∇_w^{-1}

The method of [2] for Hessian-vector multiplication begins from the Taylor expansion of the gradient about a point:

$$\nabla_w(w + \Delta w) = \nabla_w(w) + H\Delta w + O(||\Delta w||^2)$$

Where the H appears because the Hessian is the Jacobian of the gradient, assuming Schwartz symmetry of second derivatives.

Take $rv = \Delta w$ and isolate the Hessian:

$$H(rv) = rHv = \nabla_w(w + rv) - \nabla_w(w) + O(r^2)$$

$$Hv = \frac{\nabla_w(w + rv) - \nabla_w(w)}{r} + O(r)$$

Take limit as $r \to 0$:

$$Hv = \frac{\partial}{\partial r}\nabla_w(w + rv)|_{r=0}$$

Pearlmutter calculates that by defining a differential operator R (the Gâteaux derivative[5]) such that:

$$R(f(w)) = \frac{\partial}{\partial r} f(w + rv)|_{r=0}$$

and mechanically going through the equations for calculating ∇_w step by step, so that the $R(\delta)$, or the $R(\frac{J}{net})$ get cached there, too.

But, given our definition of H^{-1} above, we will actually find H^{-1} about the expansion of the functional inverse of the gradient about a point. Not the integral of the Green's function (at least not directly), not the multiplicative inverse of the gradient (unless the multiplicative inverse corresponds with the functional inverse), the functional inverse.

$$\nabla_w^{-1}(\nabla_w + \Delta w) = \nabla_w^{-1}(\nabla_w) + H^{-1}\Delta w + O(||\Delta w||^2)$$

After finding this, the entire rest of the procedure goes analogously to [2].

$$H^{-1}(rv) = rH^{-1}v = \nabla_w^{-1}(\nabla w + rv) - \nabla_w^{-1}(\nabla w) + O(r^2)$$

$$H^{-1}v = \frac{\nabla_w^{-1}(\nabla w + rv) - \nabla_w^{-1}(\nabla w)}{r} + O(r)$$

$$H^{-1}v = \frac{\partial}{\partial r}\nabla_w^{-1}(\nabla w + rv)|_{r=0}$$

And the differential operator, denoted B because of the different semantics, is as such. Note it's still just a Gâteaux derivative.

$$B(f(\nabla w)) = \frac{\partial}{\partial r} f(\nabla w + rv)|_{r=0}$$

4 Examples

4.1

$$f_1(x) = \sum_i x_i^3$$

$$\nabla f_1(x)_i = 3x_i^2$$

For reference, Hv.

$$Hv = \frac{\partial}{\partial r} 3(x_i + rv_i)^2|_{r=0}$$

$$Hv = 6x_i v_i \frac{\partial}{\partial r} 3(x_i + rv_i)^2|_{r=0}$$

Let $\nabla f_1(x)_i = z_i$

$$x_i = \sqrt{\frac{z_i}{3}}$$

Sign will actually be wrong on the result because it's really $\pm\frac{z_i}{3}$. So you do need to store the sign or switch variables to x after.

$$H^{-1}v = \frac{\partial}{\partial r}(\frac{z_i + rv_i}{3})^{\frac{1}{2}}|_{r=0}$$

$$H^{-1}v = \frac{1}{2}(\frac{z_i + rv_i}{3})^{-\frac{1}{2}}(\frac{v_i}{3})|_{r=0}$$

$$H^{-1}v = \frac{1}{2}(\frac{z_i}{3})^{-\frac{1}{2}}(\frac{v_i}{3})$$

Recall $z_i = 3x_i^2$ if you want it in terms of x.

Of course, that one was trivial anyhow. Let's do one with some off-diagonal elements in the Hessian.

4.2

$$f_2(x) = \prod_i x_i$$

Take cardinality of the x vector to be 3 to avoid extremely long derivation.

$$f_2(x) = x_1x_2x_3$$

$$\nabla f_2(x)_1 = x_2x_3$$
$$\nabla f_2(x)_2 = x_1x_3$$
$$\nabla f_2(x)_3 = x_1x_2$$

For referene, Hv.

$$(Hv)_1 = \frac{\partial}{\partial r}(x_2 + rv_2)(x_3 + rv_3)|_{r=0}$$
$$(Hv)_1 = v_2x_3 + v_3x_2 + 2rv_2v_3|_{r=0}$$
$$(Hv)_1 = v_2x_3 + v_3x_2$$
$$(Hv)_2 = v_1x_3 + v_3x_1$$
$$(Hv)_3 = v_1x_2 + v_2x_1$$

Again let $\nabla f_2(x)_i = z_i$

$$x_1 = \sqrt{\frac{z_2z_3}{z_1}}$$
$$x_2 = \sqrt{\frac{z_1z_3}{z_2}}$$
$$x_3 = \sqrt{\frac{z_1z_2}{z_3}}$$

Again, sign will actually be wrong on the result because it's really $\pm\frac{z_2z_3}{z_1}$ and so on. So you do need to store the sign or change variables.

$$(H^{-1}v)_1 = \frac{\partial}{\partial r}(\frac{(z_2 + rv_2)(z_3 + rv_3)}{(z_1 + rv_1)})^{\frac{1}{2}}|_{r=0}$$

$$(H^{-1}v)_1 = \frac{1}{2}(\frac{(z_2 + rv_2)(z_3 + rv_3)}{(z_1 + rv_1)})^{-\frac{1}{2}}\frac{(v_2(z_3 + rv_3) + v_3(z_2 + rv_2))(z_1 + rv_1) - v_1(z_2 + rv_2)(z_3 + rv_3)}{(z_1 + rv_1)^2}|_{r=0}$$

$$(H^{-1}v)_1 = \frac{1}{2}(\frac{(z_2)(z_3)}{(z_1)})^{-\frac{1}{2}}\frac{(v_2z_3+v_3z_2)(z_1)-v_1z_2z_3}{z_1^2}$$

$$(H^{-1}v)_1 = \frac{v_2x_1x_2+v_3x_1x_3-v_1x_1^2}{2x_1x_2x_3}$$

$$(H^{-1}v)_2 = \frac{v_1x_2x_1+v_3x_2x_3-v_2x_2^2}{2x_1x_2x_3}$$

$$(H^{-1}v)_3 = \frac{v_1x_1x_3+v_2x_2x_3-v_3x_3^2}{2x_1x_2x_3}$$

So you can see here that the difficult part is finding a functional inverse for the gradient, since after that the operator is completely mechanical, if rather tedious.

5 Neural Net

Of particular interest in application of Newton's method to high dimensional systems is in neural networks. I have made several attempts to attack them with this method but it is surprisingly difficult to get the functional inverse of a neural network gradient with respect to the weights which is compatible with the differential operator. Note that this is the inverse of the function from the weights to the gradients.

With minibatches (but only with very large minibatches), the functional inverse of the gradient is actually pretty trivial. However, with actual application of the differential operator, I found that unlike the DAG structure of the dependencies of the original Pearlmutter application of the operator, initial results require further results in the inverse regime.

Of course one could just approximate those numerically, so at least I present what I have right now for a 1-hidden-layer fully-connected multilayer perceptron. (You can also use the approximation of the Gâteaux derivative which doesn't take the limit of r pretty easily, but that one has numerical problems.)

The Gâteaux derivative is a formalization of the functional derivative, but I haven't gotten anywhere by using the functional derivative, although it seems tempting. Of especial note is the fact that

$$\frac{\delta f^{-1}(x)}{\delta f(y)} = -\frac{\delta(f^{-1}(x)-y)}{f'(f^{-1}(x)}$$

But then that doesn't help the dependency problem. The same holds for finding the Legendre transformation.

Often the Jacobian determinant is zero, but you can avoid this by construction (pick another random initialization). I assume you are completely familiar with neural nets already[3].

Specific example is with sum of squared error and no softmax.

∇_w of neural net:

$$net_1 = xW_1$$

$$h_1 = act(net_1)$$

$$net_2 = h_1W_2$$

$$h_2 = act(net_2)$$

$$J = err(h_2, y)$$

$$\frac{\partial J}{\partial h_2} = h_2 - y$$

$$\frac{\partial J}{\partial net_2} = act'(net_2)\frac{\partial J}{\partial h_2}$$

$$\frac{\partial J}{\partial h_1} = \frac{\partial J}{\partial net_2}W_2^T$$

$$\frac{\partial J}{\partial net_1} = act'(net_1)\frac{\partial J}{\partial h_1}$$

$$\frac{\partial J}{\partial W_2} = \frac{\partial J}{\partial net_2}h_1$$

$$\frac{\partial J}{\partial W_1} = \frac{\partial J}{\partial net_1} x$$

For reference, here is also listed the steps to get Hv:

$$R(net_1) = R(x)W_1 + xv_1$$
$$R(h_1) = act'(net_1)R(net_1)$$
$$R(net_2) = R(h_1)W_2 + h_1 v_2$$
$$R(h_2) = act'(net_2)R(net_2)$$
$$R(\frac{\partial J}{\partial h_2}) = R(h_2)$$
$$R(\frac{\partial J}{\partial net_2}) = R(\frac{\partial J}{\partial h_2})act'(net_2) + \frac{\partial J}{\partial h_2}act''(net_2)R(net_2)$$
$$R(\frac{\partial J}{\partial h_1}) = R(\frac{\partial J}{\partial net_2})W_2^T + \frac{\partial J}{\partial net_2}v_2^T$$
$$R(\frac{\partial J}{\partial net_1}) = R(\frac{\partial J}{\partial h_1})act'(net_1) + \frac{\partial J}{\partial h_1}act''(net_1)R(net_1)$$
$$Hv_2 = R(\frac{\partial J}{\partial W_2}) = R(\frac{\partial J}{\partial net_2})h_1 + \frac{\partial J}{\partial net_2}R(h_1)$$
$$Hv_1 = R(\frac{\partial J}{\partial W_1}) = R(\frac{\partial J}{\partial net_1})x + \frac{\partial J}{\partial net_1}R(x)$$

The functional inverse of the gradient. Note that this requires pseudoinverse usage (or chiral inverse). There are actually quite a few inverses possible, all of which have many conditions just as the pseudoinverse does and all of the ones I've investigated have this dependency loop problem.

$$\frac{\partial J}{\partial net_1} = \frac{\partial J}{\partial W_1}x^+$$
$$\frac{\partial J}{\partial h_1} = \frac{\partial J}{\partial net_1}act'^{-1}(net_1)$$
$$\frac{\partial J}{\partial net_2} = \frac{\partial J}{\partial h_1}W_2^{-T}$$
$$\frac{\partial J}{\partial h_2} = \frac{\partial J}{\partial net_2}act'^{-1}(net_2)$$
$$h_2 = \frac{\partial J}{\partial h_2} + y$$
$$net_2 = act^{-1}(h_2)$$
$$h_1 = net_2 W_2^{-1}$$
$$net_1 = act^{-1}(h_1)$$
$$W_2 = h_1^+ net_2$$
$$W_1 = x^+ net_1$$

Finally, what would be $H^{-1}v$:

$$B(\frac{\partial J}{\partial net_1}) = v_1 x^+ + \frac{\partial J}{\partial W_1}B(x^+)$$
$$B(\frac{\partial J}{\partial h_1}) = B(\frac{\partial J}{\partial net_1})act'^{-1}(net_1)\frac{\partial J}{\partial net_1}act''^{-1}(net_1)B(net_1)$$

Note, as mentioned, that we don't actually have $B(net_1)$ yet.

$$B(\frac{\partial J}{\partial net_2}) = B(\frac{\partial J}{\partial h_1})W_2^{-T} + \frac{\partial J}{\partial h_1}B(W_2^{-T})$$

$$B(\frac{\partial J}{\partial h_2}) = B(\frac{\partial J}{\partial net_2})act'^{-1}(net_2)\frac{\partial J}{\partial net_2}act''^{-1}(net_2)B(net_2)$$

$$B(h_2) = B(\frac{\partial J}{\partial h_2})$$

$$B(net_2) = (act^{-1})'(h_2)B(h_2)$$

$$B(h_1) = B(net_2)W_2^{-1} + net_2 B(W_2^{-1})$$

$$B(net_1) = (act^{-1})'(h_1)B(h_1)$$

$$H^{-1}v_2 = B(W_2) = B(h_1^+)net_2 + h^+ B(net_2)$$

$$H^{-1}v_1 = B(W_1) = B(x^+)net_1 + x^+ B(net_1)$$

At this point, one understands why Pearlmutter used a variant short notation.

It may be more worth the while to find a network architecture where the weight-to-gradient function is involutive, or at least less of a pain to invert in a way compatible with the differential operator. I have begun work on this. The situation reminds me of the situation with loopy BP.

6 Acknowledgements

Thanks to the CSPAM peeps, HG, JB, JH and SCM for letting me annoy them while writing this. Preemptive apologies to JLM and JB for doing weird things. Any mistakes are my fault.

References

[1] Lang, S. (1995). Differential and Riemannian manifolds, Vol. 160 of. Graduate Texts in Mathematics, 185.

[2] Pearlmutter, B. A. (1994). Fast exact multiplication by the Hessian. Neural computation, 6(1), 147-160.

[3] Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). Deep learning (Vol. 1). Cambridge: MIT press.

[4] Martens, J. (2010, June). Deep learning via Hessian-free optimization. In ICML (Vol. 27, pp. 735-742).

[5] Gâteaux, R. (1913). Sur les fonctionnelles continues et les fonctionnelles analytiques. CR Acad. Sci. Paris, 157(325-327), 65.

[6] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.