NOTE **Communicated by Nicol Schraudolph**

Efficient Calculation of the Gauss-Newton Approximation of the Hessian Matrix in Neural Networks

Michael Fairbank
michael.fairbank1@city.ac.uk
Eduardo Alonso
E.Alonso@city.ac.uk
Department of Computing, School of Informatics, City University London, London EC1V 0HB, U.K.

The Levenberg-Marquardt (LM) learning algorithm is a popular algorithm for training neural networks; however, for large neural networks, it becomes prohibitively expensive in terms of running time and memory requirements. The most time-critical step of the algorithm is the calculation of the Gauss-Newton matrix, which is formed by multiplying two large Jacobian matrices together. We propose a method that uses backpropagation to reduce the time of this matrix-matrix multiplication. This reduces the overall asymptotic running time of the LM algorithm by a factor of the order of the number of output nodes in the neural network.

1 Introduction

A neural network is a smooth function $\vec{y} = \vec{y}(\vec{x}, \vec{w})$ that maps an input column vector $\vec{x}$ to an output column vector $\vec{y}$ and where $\vec{w}$ is a parameter vector known as the weight vector.

For the specific input and output vectors $\vec{x}_p$ and $\vec{y}_p$, corresponding to a training pattern p, the Jacobian matrix of the neural network is defined to be $J_p = \frac{\partial \vec{y}_p}{\partial \vec{w}}$, which is a matrix with element (i, j) equal to $\frac{\partial (\vec{y}_p)^i}{\partial (\vec{w})^j}$. The Gauss-Newton matrix is defined to be $G = \sum_p G_p$, where $G_p = {J_p}^T J_p$. We define $n_w = \dim(\vec{w})$, $n_o = \dim(\vec{y})$ and n_p as the number of training patterns. Then J_p is a $n_o \times n_w$ matrix, and so forming the matrix G by direct matrix multiplication and summation over all patterns would take $2 n_o n_p {n_w}^2$ floating point operations (flops), ignoring lower power terms.

We define a technique that can calculate the G matrix in the faster time of approximately $3 n_p {n_w}^2$ flops (ignoring lower-power terms). This faster algorithm is related to the method of Schraudolph (2002) and exploits a trick that backpropagation (Werbos, 1974; Rumelhart, Hinton, & Williams, 1986) can be used to quickly multiply an arbitrary column vector on the left by ${J_p}^T$.

Forming the G matrix is important because it is central to the Levenberg-Marquardt (LM) training algorithm (Levenberg, 1944; Marquardt, 1963). The LM algorithm uses a weight update that requires the inverse of G. Details are given by Bishop (1995). Since $G \in \Re^{n_w \times n_w}$, the inversion of G will take time $O({n_w}^3)$, and since usually $n_p \gg n_w$, it turns out that the formation of the matrix G is usually slower than its inversion. Hence, our algorithm is reducing the asymptotic time of the most time-critical step of the LM algorithm. Previous research to speed up the formation of G has concentrated on parallel implementations (Suri, Deodhare, & Nagabhushan, 2002).

2 The Technique

Backpropagation is an algorithm to calculate the gradient $\frac{\partial E_p}{\partial \vec{w}}$ very efficiently for a given pattern p and error function, E_p. If we assume the computations at the nodes of the network are dwarfed by those at the network weights, then the backpropagation algorithm takes $3n_w$ flops per pattern.

By the chain rule, $\frac{\partial E_p}{\partial \vec{w}} = \frac{\partial \vec{y}}{\partial \vec{w}}^T \frac{\partial E_p}{\partial \vec{y}} = {J_p}^T \frac{\partial E_p}{\partial \vec{y}}$. Hence, we see that backpropagation can be used to multiply a column vector, $\frac{\partial E_p}{\partial \vec{y}}$, very efficiently on the left by the transposed Jacobian matrix. The choice of column vector here is arbitrary; it does not have to specifically be $\frac{\partial E_p}{\partial \vec{y}}$. This is the trick we use to create our fast algorithm for calculating G.

A standard method to calculate the Jacobian matrix is as follows. To calculate the ith row of J_p, we use backpropagation to multiply ${J_p}^T$ by the ith column of I, an $n_o \times n_o$ identity matrix. Repeating this for all $i \in \{1, 2, \ldots, n_o\}$ outputs will calculate the full J_p matrix in $3n_o n_w$ flops.

The new method to calculate the G_p matrix is as follows. Since $G_p = {J_p}^T J_p$, the ith column of G_p is equal to the product of the matrix ${J_p}^T$ with the ith column of J_p. Hence each column of G_p can be calculated using one pass of backpropagation. Therefore, calculating the whole G_p matrix from a given J_p matrix takes $3{n_w}^2$ flops.

In addition to the time taken to calculate J_p and G_p, we also need one initial forward pass through the network, which will take $2n_w$ flops. Hence, the total flop count to calculate G, when summing over all n_p patterns, is $n_p\left(2n_w + 3n_o n_w + 3{n_w}^2\right)$. Since usually $n_o \leq \sqrt{n_w}$, the most significant term here is $3n_p{n_w}^2$ flops.

3 Discussion

Since the work of Schraudolph (2002) allows fast multiplication of the G matrix by an arbitrary column vector, in time $7n_p n_w$ flops, it would be trivial to extend that work to form the full G matrix column by column.

This would give an asymptotically equivalent algorithm to ours, but in a slower absolute flop count of $7n_p{n_w}^2$.

The calculation time of the direct multiplication method and our method could both be halved further by exploiting the symmetry of G.

Our calculations indicate that while Strassen multiplication (Huss-Lederman, Jacobson, Tsao, Turnbull, & Johnson, 1996) is not useful in calculating G_p for a single pattern, it does confer an asymptotic advantage when calculating G for all patterns in a single outer product. However, doing so is memory intensive and significantly more complicated to implement than our method.

We have not considered hardware acceleration and caching issues, both of which would likely favor conventional matrix multiplication over our method.

4 Conclusion

We have presented a way to use backpropagation to reduce the time taken to calculate the Gauss-Newton matrix in Levenberg-Marquardt down by a factor proportional to n_o. This reduces the critical time step in implementing the LM algorithm and so could be a useful tool to optimize any LM implementation where $n_o \gg 1$.

Acknowledgments

We are grateful to the anonymous reviewers for their suggestions for this note.

References

Bishop, C. M. (1995). *Neural networks for pattern recognition*. New York: Oxford University Press.

Huss-Lederman, S., Jacobson, E. M., Tsao, A., Turnbull, T., & Johnson, J. R. (1996). Implementation of Strassen's algorithm for matrix multiplication. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (CDROM), Supercomputing '96*. Washington, DC: IEEE Computer Society.

Levenberg, K. (1944). A method for the solution of certain non-linear problems in least squares. *Quart. Appl. Math.*, *2*, 164–168.

Marquardt, D. W. (1963). An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, *11*, 431–441.

Rumelhart, D., Hinton, G., & Williams, R. (1986). Learning representations by back-propagating errors. *Nature*, *6088*, 533–536.

Schraudolph, N. N. (2002). Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, *14*(7), 1723–1738.

Suri, N.N.R.R., Deodhare, D., & Nagabhushan, P. (2002). Parallel Levenberg-Marquardt-based neural network training on Linux clusters—a case study. In *Linux Clusters, ICVGIP 2002, 3rd Indian Conference on Computer Vision, Graphics and Image Processing*. N.P.: Allied Publishers.

Werbos, P. J. (1974). *Beyond regression: New tools for prediction and analysis in the behavioral sciences*. Unpublished doctoral dissertation, Harvard University.

Received February 25, 2011; accepted September 16, 2011.