MSc Neural Networks: Coursework 2

John Downing

Motivation for Newton's Method

Classical backpropagation uses a linear approximation to the error function – the gradient – in order to iteratively proceed towards a minimum on the error surface. The gradient is just the derivative of the error with respect to the function parameters (the network weights), and each weight is simply updated in proportion to its contribution to the gradient at the current point on the error surface. If the function that the network is trying to approximate is not a straight line/plane/hyperplane, then this is likely to be inefficient- resulting in a "zig-zag" trajectory towards the minimum (Haykin, 2009). More advanced methods of optimisation use higher-order information about the error surface, in order to make better predictions about the location of the nearest minimum – and consequently converge in less steps.

As Bishop (1995, p.141) points out, backpropagation itself is not so much a training algorithm as a "computationally efficient method for evaluating … derivatives". Backpropagation can, then, also be used to calculate e.g. the second order derivatives of the error with respect to the weights – the Hessian – which allows a quadratic approximation to the error function at a point on the surface. Netwon's Method makes use of the Hessian in order to determine the direction of a point where the *derivative* of the error function is zero (https://youtu.be/28BMpgxn_Ec). However, calculation of the Hessian can be computationally intensive – and in addition has strict mathematical requirements (in that the matrix much be "positive definite at each iteration of the algorithm" (Haykin, 2009; p.98)). As a consequence, a number of techniques have been developed which allow a more practical approximation of the Hessian to be calculated.

Levenberg-Marquardt Algorithm

The Levenberg-Marquardt algorithm trades off the benefits of Newton's Method (rapid convergence) with those of gradient descent (guaranteed convergence, given appropriate learning rate) – since Newton's Method by itself is not assured of convergence (Haykin, 2009). The algorithm can make use of the exact Hessian, but typically works with an approximation generated from first order derivatives: the outer product of the *Jacobian* with itself.

The Jacobian is the derivative of the network *output* with respect to the weights[1]. This is (implicitly) calculated during classical backpropagation – so the procedure just needs to be modified to tease these intermediate vectors[2] out from the calculation of the gradients. Following the derivations provided at http://homepages.gold.ac.uk/nikolaev/311bpr.htm, it can be seen that the partial derivatives of the network error, with respect to each weight, are of the form:

$$\delta error/\delta weights = \delta error/\delta output * \delta output/\delta weights$$
$$= \delta error/\delta output * Jacobian$$

By fully expanding the chain rules for the derivatives at each layer of the network, we can see which components of the chain relate to the node outputs, and which relate to the network error:

For weights j→k on connections to nodes in the (unthresholded) output layer (with a single output node)

$$\delta error_k/\delta w_{jk} = \delta error_k/\delta output_k * \delta output_k/dsum_k * \delta sum_k/\delta w_{jk}$$
$$= \delta error_k/\delta output_k * \delta output_k/\delta sum_k * output_j$$
$$= -(yk - ok) * 1 * output_j$$

For weights i→j on connections to nodes in the (sigmoidal) hidden layer (with a single output node)

$$\delta error_k/\delta w_{ij} = \delta error_k/\delta output_k * \delta output_k/\delta sum_k * \delta sum_k/\delta output_j * \delta output_j/\delta sum_j * \delta sum_j/\delta w_{ij}$$
$$= \delta error_k/\delta output_k * \delta output_k/\delta sum_k * \delta sum_k/\delta output_j * \delta output_j/\delta sum_j * x_i$$
$$= \delta error_k/\delta output_k * \delta output_k/\delta sum_k * \delta sum_k/\delta output_j * out_j(1-out_j) * x_i$$
$$= \delta error_k/\delta output_k * \delta output_k/\delta sum_k * w_{jk} * out_j(1-out_j) * x_i$$
$$= -(yk - ok) * 1 * w_{jk} * out_j(1-out_j) * x_i$$

The Jacobian vector is summed across all input patterns, and then divided by N, the number of patterns, so that the approximate hessian is given by $1/N * J * J^T$. A regularisation value (e.g. 0.001) is added to the diagonals of the approximate Hessian, so that it can definitely be inverted – and then its inverse is multiplied by the gradient vector to give the weight updates for the epoch (Haykin, 2009).

---

[1] Annoyingly, Bishop and http://www.cs.utah.edu/~tch/notes/matlab/ReBEL/code/ReBEL-0.2.5/netlab/mlpjacob.m talk about the Jacobian being the output with respect to each of the *inputs*.

[2] For a single input pattern, the Jacobian is a vector of length w, where w is the number of weights in the network.

Notes on accompanying code

I actually developed two models for this assignment, the first of which (submitted as `assign2.m`) was based on an "object oriented" approach that I nearly used for the previous coursework. Although a batch rather than online design, out of necessity (for calculating the Jacobian), this model still used loops to iterate over each pattern – and simply kept a running total of the gradient vector, and approximate hessian, as it went along. Although successful as a proof of concept – and correctness – using the "simple neural network" specified in the requirements, this approach did not scale well to the fully connected, 10 input / 5 hidden node network, intended for use with the sunspots data. It was, in fact, unbelievably slow (!)

My second attempt (`assign2b.m`) therefore reverted to a matrix approach, starting out as a modified version of MLPts.m from the first term[3]. However, I struggled to construe my chain rule derivation of the Jacobian in terms of matrix algebra – and so ultimately arrived at a hybrid approach, using a same-sized matrix for each element in the chain (e.g. 50 x 200, for weights to the input layer) and then just multiplying cell-by-cell to get the values for each pattern, and explicitly summing across rows where necessary. This is obviously not an ideal solution, but is quick – and did allow me to verify my calculations at each stage in the chain against those from the earlier model[4].

[3] http://homepages.gold.ac.uk/nikolaev/MLPts.m

[4] In fact, the matrix calculations were mostly reverse-engineered to match those from the earlier model.

References

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

Haykin, S. (2009). *Neural networks and learning machines*. Upper Saddle River: Prentice-Hall.