A Comparison of Back Propagation Implementations

Jondarr Gibb and Leonard Hamey
Department of Computing
Macquarie University

Technical Report C/TR95-06
September, 1995

Abstract
Back propagation training algorithms have been implemented by many researchers for their own purposes and provided publicly on the internet for others to use in verification of published results and for reuse in unrelated research projects. Often, the source code of a package is used as the basis for a new package for demonstrating new algorithm variations, or some functionality is added specifically for analysis of results. However, there are rarely any guarantees that the original implementation is faithful to the algorithm it represents, or that its code is bug free or accurate. This report attempts to look at a few implementations and provide a test suite which shows deficiencies in some software available which the average researcher may not be aware of, and may not have the time to discover on their own. This test suite may then be used to test the correctness of new packages.

1 Introduction

There are many neural network packages freely available on the Internet which allow the investigator to reuse an implementation developed and tested elsewhere. However, what guarantees are there that such an implementation is a working package? Many researchers in the area seem to have come to the conclusion that it is more worthwhile to develop an in-house package for the specific needs of the research at hand, and then make that package available when it has been used for some time, and its usefulness has been proven. If everyone follows this policy, then we have a proliferation of public packages which no-one else can use, some of which are not tested beyond the application for which they were written. Each of these packages will have some useful features, but may not be perfectly suited to an average researcher’s task at hand (and may therefore need to be further modified by the researcher).

Not only is it imperative that a researcher know if a public domain package is correctly implementing the desired algorithm, is accurate, and is flexible enough to do what is required, it is also important that any package developed in-house be verifiable before it is used to demonstrate results in comparative studies and support arguments for newly-developed algorithms.

This report attempts to codify a series of tests for the researcher/developer to use on back propagation simulation packages of any origin, including pitfalls for the developer. We also look at a few commonly used and freely available packages.

In addition, the way in which implementation variations may affect results is implied by the application of these tests. Whether such effects are within the tolerance of experimental error, or are statistically insignificant in the realm of Neural Network simulations is an important question. Kolen and Pollack [7] have already investigated the effects of initial conditions (that is, weight configuration) on the outcome of experiments, and we intend to show how implementation differences may likewise have an effect on the final outcome of training a network. Xie and Jabri [26], and also Sakaue, et al [17] have investigated low-precision networks (around 10 bits), and found that the
standard back propagation algorithm simply does not work reliably with so few bits unless some modification is made. The authors’ experience with 16-bit real number encoding (see section 3.9) adds to the argument that full real-number precision is required.

If accuracy is an important factor in implementing an algorithm, due to the convolutions of error-weight space, and neural networks are a stable platform from which artificially intelligent applications may be built, then precise implementations must be used if the same results are to be obtained by a given network architecture and weight configuration no matter which package is used. However, if the non-determinism of a semi-chaotic system, where small deviations to the adjustment of weights (dependent on implementation and other non-reproducible factors) is desirable, then this test suite has no relevance.

A study conducted by Prechelt [12] confirms the generally unscientific nature of neural network research, and this paper’s intention is to try and deal with the most fundamental part of testing – checking an implementation before the algorithm represented can be used in any comparison.

The authors of this report are of the opinion that reproducibility of results without the use of statistical analysis and approximation over many network train/test runs is important, and we therefore recommend such implementation verification techniques to you.

2 The Test Suite

The test suite has been developed to test many variations on the standard back propagation algorithm. Each test should reveal a specific area of possible incorrectness in an implementation’s handling of an algorithm variation.

The XOR network was chosen as a very simple test which still explores the possibilities of poor training. Both Kuyper [19, 18] and Hamey [6, 5] have shown that multiple global minima and saddle points exist within the error surface, making it a non-trivial problem to find the error minimum of the network. Most simulation packages which claim to implement back propagation do provide an example XOR network, which makes it the easiest network to test, and would imply that the developer of that package has already tested it.

Of course, historically, the XOR network was used by Minsky [10] to disprove the viability of perceptrons, and then investigated thoroughly by the likes of Rumelhart, et al [15] [16] to prove the usefulness of the back propagation algorithm in multi-layer perceptron networks.

Given an initial weight configuration as shown in Figure 1 for the XOR network, and input/output patterns as described in Table 1, a series of tests were devised to examine the opera-

Figure 1: The exclusive-or network used for testing.
\[
\begin{array}{c|ccc}
 & u_1 & u_2 & t \\
\hline
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 \\
\end{array}
\]

Table 1: Training Data for Exclusive-or Task

\[
\begin{array}{c|ccc}
 & \text{Initial} & BP_1 & BP_{300} \\
\hline
a & -0.5 & -0.51912587266650 & -1.892107259295294 \\
b & -1.2 & -1.21596398690630 & -3.58307904796538 \\
c & 1.0 & 0.99922612812898 & 3.31725936197670 \\
d & -2.0 & -2.01613689525158 & -2.8568490476723 \\
e & 4.0 & 3.99411550305494 & 4.78245510423537 \\
f & -4.0 & -4.0097488783368 & -5.03232137699571 \\
g & -1.0 & -1.03475043628124 & -2.36167416634203 \\
h & 2.0 & 1.99527672350250 & 4.83916744885057 \\
j & 2.0 & 2.00561233895710 & 4.73132260757211 \\
\end{array}
\]

Table 2: Weights and network responses – initially, plain back propagation.

...uction of an implementation of a back propagation algorithm with some standard variations. These weights were chosen to divide the XOR problem such that one of the hidden nodes will react positively to one of the high-output test patterns \((1,0)\), while the other hidden node reacts to the other pattern \((0,1)\).

The initial responses of the network, using an asymmetric sigmoid function \(f(x) = \frac{1}{1 + e^{-x}}\), after the weights were so initialised, are shown in the first column of table 2.

If a network does not give these results, then there are two possible conclusions that can be drawn. The first possibility is that the network does not perform the feed-forward operations successfully. This would show up mostly in the fourth pattern, and may show very little variation in the first pattern. The accumulation of inputs multiplied by weights at each node, or application of the transfer function may be in error.

The second possibility is that the weights are incorrectly initialised, or else the operation to set the weights as specified is poorly implemented in the package or misunderstood by the user. If there is an error in the first pattern, then the bias weight(s) (the only ones truly contributing to the output error) are those most likely to be incorrect.

If the values at the output are closer to 0 for all cases, then symmetric neurons (see section 2.5) may be in use.

2.1 Plain Back Propagation

The first test of an implementation is to make sure that back propagating simple errors through the network achieves the expected results. The second column of table 2 \((BP_1)\) shows the outputs and weights after running a network for one epoch with a learning rate of 0.4. Training is done in batch mode – that is, all patterns were presented before the weights were updated.
<table>
<thead>
<tr>
<th></th>
<th>(BPM_2)</th>
<th>(BPM_{300})</th>
<th>(BPD_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>-0.54904361534396</td>
<td>-2.50873891921042</td>
<td>-0.53154011240917</td>
</tr>
<tr>
<td>(b)</td>
<td>-1.24195178195313</td>
<td>-4.65594977766704</td>
<td>-1.22108237803349</td>
</tr>
<tr>
<td>(c)</td>
<td>0.99962130229960</td>
<td>4.42302400843016</td>
<td>0.99215166218877</td>
</tr>
<tr>
<td>(d)</td>
<td>-2.0423568291767</td>
<td>-3.0982151693036</td>
<td>-2.01507979182334</td>
</tr>
<tr>
<td>(e)</td>
<td>3.98496389527666</td>
<td>5.42300184786244</td>
<td>3.95716437217267</td>
</tr>
<tr>
<td>(f)</td>
<td>-4.02577733390025</td>
<td>-5.59111928053655</td>
<td>-3.9870323074045</td>
</tr>
<tr>
<td>(g)</td>
<td>-1.0885159948910</td>
<td>-3.24593368313184</td>
<td>-1.05607598692301</td>
</tr>
<tr>
<td>(h)</td>
<td>1.98955495522621</td>
<td>6.69557008476025</td>
<td>1.9768492967697</td>
</tr>
<tr>
<td>(j)</td>
<td>2.01662839649425</td>
<td>6.47955345642534</td>
<td>1.9966873756947</td>
</tr>
</tbody>
</table>

Table 3: Weights and network responses – momentum and weight decay.

The reason for using a learning rate of 0.4 is that a value of, say, 1 would give results indistinguishable from an implementation which does not allow varying of the learning rate, or implements it incorrectly. A value of 0.5 may give a similarly prejudiced set of values (as noted later).

These results are verified values to fourteen decimal places over at least two of the packages tested, and confirmed to fewer places by other packages.

Any variation to these values would indicate a problem in the back propagation of errors (given that the initial responses were correct). The possible causes of variations from these values include the incorrect application of the derivative of the sigmoid function, a variation on the output error function such as back propagating the hyperbolic tangent of the difference between target and actual values (see section 2.4), incorrect accumulation of errors at each node (through node-based accumulation, instead of weight-based), and incorrect use, or non-use, of bias values (such as in the bias update rule).

There are also major implementation variations which may be discovered at this point – for instance, if these results are achieved with a learning rate of 0.2, instead of 0.4, then a design decision was made to remove the term of \(\frac{1}{2}\) in the calculation of error at a given node, as specified by Rumelhart, et al [15].

A deviation from the figures may also occur if on-line training is being performed instead of batch. See section 2.6 to compare the results. As a more basic test of the network, cut down the training set to consist of only the first pattern, and compare results to those shown in table 7.

The third column (\(BP_{300}\)) shows the weights and outputs after running the same network for 300 epochs. The purpose of this test is to show up any accumulated error in accuracy in the calculation of errors over many epochs. Variations will occur here where the real numbers used do not have sufficient precision to follow the same path in error-weight space.

### 2.2 Momentum and Weight Decay

One of the most common variations to the back propagation algorithm is the addition of a momentum term. Momentum should speed up the learning of most problems. The first column of table 3 (\(BPM_2\)) shows the outputs and weights for the network after running for two epochs with a learning rate of 0.4 and a momentum term of 0.7.

The value chosen for testing the momentum should not be a multiple of the learning rate, and should also not be 1. Although this value is not optimal, it tends to show up pitfalls in software development.

The network must be run for two epochs to show up the effect of momentum (to incorporate a previous weight change). If there is any variation to the above figures, then the additional term
\[
\begin{array}{|c|c|c|c|}
\hline
 & BP_{\sigma_1} & QP_2 & QP_{30} \\
\hline
a & -0.53522647807613 & -0.5691258164379 & -4.8238498734016 \\
b & -1.22909731552026 & -1.25871396820475 & -15.7826357374020 \\
c & 0.99785627889232 & 0.99964993397670 & 9.6592110302038 \\
d & -2.03128232786116 & -2.05930042293184 & -4.7378466343633 \\
e & 3.98489932003890 & 3.97878717347490 & 10.4789166117146 \\
f & -4.01482391617252 & -4.03601365667561 & -17.11487090041490 \\
g & -1.0459798168148 & -1.12500395298630 & -5.94134554104835 \\
h & 1.99391841640223 & 1.98791760276279 & 12.83799891140993 \\
j & 2.01000679775595 & 2.02252135239921 & 12.04819847516085 \\
\hline
\end{array}
\]

Table 4: Weights and network responses – sigmoid prime offset and quickprop.

of the momentum constant multiplied by the previous weight change of a given node \((a \Delta w_{t-1})\) is not correctly implemented. This may be due to the fact that momentum is not applied to bias nodes, for instance, or because a non-standard interpretation of the weight update formula has been implemented.

Any accumulated error over 300 epochs will be evident by comparing with the second column of table 3 \((BP_{300})\). Although accuracy over time has already been tested, it is important to make sure that momentum adds no further complexity beyond a lack of precision in real numbers. A variation may also occur from the values given if some of the data are not initialised properly (such as previous weight change). This may cause a different path in error-weight space to be chosen, the effects of which become obvious only after many epochs.

Another major variation to the back propagation algorithm is the introduction of weight decay, which is a form of regularisation whereby each weight in the network is decreased by a small percentage at each update so as to limit the growth of weights, hopefully shrinking small (unused) weights to zero.

The third column of table 3 \((BP_{D_2})\) shows the effect of a weight decay of 0.01 (which would normally be considered an extraordinarily high value) on back propagation learning with a learning rate of 0.4. This value was chosen so that the effects of weight decay can be seen quickly.

Variations from these values would indicate that weight decay is not being consistently applied to weights in the network. Decay should be applied to the current weight before it is updated. A possible problem is that no decay is being applied to the bias weights. The sign of the decay term should also be checked to see if ‘decay’ occurs with a positive-valued term if appropriate.

### 2.3 Sigmoid Prime Offset and Quickprop

There were many developments related to the back propagation algorithm introduced by Fahlman [4], including a small offset value used to modify the shape of the inverse of the sigmoid transfer function, and the second-order Quickprop algorithm.

Anyone implementing these variations based on Fahlman’s paper should compare their results with those shown in table 4.

The first column of the table \((BP_{\sigma_1})\) shows the results of running a network with a learning rate of 0.4 and a sigmoid prime offset of 0.1 for one epoch.

The second column \((QP_2)\) shows learning with the quickprop algorithm for two epochs with a learning rate of 0.4 and a maximal weight growth factor of 1.75 (as suggested by Fahlman). The network must be run for two epochs to show the effects of the second order term. For most
packages directly related to Fahlman’s original software (see section 3.4), a Mode Switch Threshold of 0 might be required to perform quickprop.

The third column, table 4 ($Q_P3o$), shows the result of running quickprop for 30 epochs, and will indicate any accumulative error in an implementation, and may also show up the implementation decision on the use of the maximal weight growth factor (as discussed in section 3.4). If the path in error-weight space is different, then some variables in the implementation may not have been initialised (such as the previous derivative).

### 2.4 Split Eta, Hyperbolic Error and Don’t Care Regions

The first column of table 5 ($BPSE_1$) shows the result of training the network for one epoch with a learning rate of 0.4 which is split. Split Eta (also known as Split Epsilon) is a method by which the learning rate used to train any node is the network learning rate constant divided by the number of weights which contribute to the error at that node. For example, each of the hidden layer nodes in figure 1 have three weights (from the two input nodes, plus the bias), and therefore the network learning rate would be divided by three. Similarly, the output node’s learning rate would also be one third of that applied to the network.

Application of this method ‘evens out’ the effect of larger input sums to nodes with more weights in the case where this might vary across the layers of a network, or even within a layer. This test checks that the implementation is correct – as all nodes will have the same learning rate in this case, the results should also be compared with a learning rate of 0.4/3 without split eta.

The second column of table 5 ($BPH_1$) shows the result of applying a hyperbolic arctangent to the sum of squared errors at the output node of the network. This method results in a smoother output error before propagating it to lower layers of the network. It is important that this is applied to the output layer for each pattern before the error back propagation occurs (and is only applied to the output layer).

$$\text{Error}_p = f'(x_p) * \text{arctanh(desired\_output}_p - x_p)$$

The third column ($BPDC_{300}$) shows the result of adding Don’t care regions to the network which is trained with a learning rate of 0.4 and momentum of 0.7. These results should be compared with table 3 ($BPM_{300}$). The regions were implemented such that any error at the output node between −0.1 and 0.1 would be considered a 0.

The reason for using momentum in this test is to specifically show the difference when Don’t care regions are used. Backpropagation without momentum does not show any significant variation.
Table 6: Weights and network responses – symmetric nodes.

<table>
<thead>
<tr>
<th></th>
<th>Initial</th>
<th>$BPSym_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>-0.5</td>
<td>-0.44043772755484</td>
</tr>
<tr>
<td>$b$</td>
<td>-1.2</td>
<td>-1.18391332452406</td>
</tr>
<tr>
<td>$c$</td>
<td>1.0</td>
<td>1.04448166080768</td>
</tr>
<tr>
<td>$d$</td>
<td>-2.0</td>
<td>-1.98693590116876</td>
</tr>
<tr>
<td>$e$</td>
<td>4.0</td>
<td>4.01257440901559</td>
</tr>
<tr>
<td>$f$</td>
<td>-4.0</td>
<td>-3.99903183251147</td>
</tr>
<tr>
<td>$g$</td>
<td>1.0</td>
<td>1.15238160539653</td>
</tr>
<tr>
<td>$h$</td>
<td>2.0</td>
<td>1.99102932574011</td>
</tr>
<tr>
<td>$j$</td>
<td>2.0</td>
<td>1.97494125852025</td>
</tr>
<tr>
<td>$r(0, 0)$</td>
<td>-0.00162819883462</td>
<td>0.04668799567503</td>
</tr>
<tr>
<td>$r(0, 1)$</td>
<td>0.06214300389929</td>
<td>0.11341977196344</td>
</tr>
<tr>
<td>$r(1, 0)$</td>
<td>0.24469668381365</td>
<td>0.27589772057647</td>
</tr>
<tr>
<td>$r(1, 1)$</td>
<td>-0.02447285387248</td>
<td>0.03123264110705</td>
</tr>
</tbody>
</table>

over 300 epochs, so there must be a decision made when travelling in error-weight space only when momentum is applied which is critical to the functioning of the network.

2.5 Symmetric Neurons

All of the discussion and testing so far has been based on asymmetric nodes, having an output value between 0 and 1, however, symmetric nodes are often used when the input patterns are centred around 0.

For a transfer function defined by $f(x) = \frac{1}{1+e^{-x}} - 0.5$, and a small change in the weights used to initialise the network (shown in table 6, changing weight $g$ only), the initial responses of the network are given. The results after running the network for one epoch with a learning rate of 0.4 is shown in table 6 column $BPSym_1$. There is no need to modify the input/output patterns for this test.

Any variation from these values, if no other test has shown an anomaly, would indicate that either the transfer or inverse transfer function was incorrectly implemented.

2.6 On-line Training

As a complete test, and for general reference, table 7 column $BP_{P10}$ shows the result of updating the weights of the network after presenting only the first pattern. This can be achieved by training for one epoch on a training set consisting of only that pattern. As can be seen, only the weights associated with the bias nodes are updated, and this column can therefore be used to verify the correct implementation of bias nodes – whether the package used is restricted to one of batch or on-line processing, or not.

The next column ($BP_{P11}$) gives the weights after the introduction of the next pattern, which shows the network’s response to a non-zero input pattern. If the bias nodes are correctly implemented, then this column’s results will indicate the bare minimum correct interpretation of on-line training.

All of the previous sections have discussed batch training. Table 7 columns $BPO_{E1}$ and $BPO_{E2}$ show the results of training a network with a learning rate of 0.4 and on-line training for one and two epochs respectively. An epoch is defined as the presentation of all (four) training patterns; however, when using on-line training, weights are updated after each pattern is presented to the network, not at the end of the epoch.

In most simulation packages which primarily support batch operations, on-line training is achieved through the use of a built-in switch, or a simple change in the source code to move the
weight update procedure. However, it is not necessarily easy to achieve batch training with a package which is dedicated to an on-line operation. Some applications do require only on-line training, and because of this, the above results are given to test this feature.

A deviation from the above results may be caused by an improper placement of the update procedure in the source code.

3 Simulation Packages

There are many freely available packages on the Internet, and although none of them claim to be perfect, there is an expectation that anything made available in this manner should be reliable. Similarly, although it is never explicitly stated, a researcher will expect that the algorithm claimed for a given implementation is carried through faithfully. However, this is rarely the case. The following simulation packages had all claimed that they implemented the back propagation algorithm.

3.1 Matlab Neural Network Toolbox

Matlab\(^1\) is the only commercial package tested in this report, and this made it stand out somewhat through its professional presentation of documentation, its user support, and its ease of use (for those who are used to Matlab-related products).

The high-precision output mode was used to verify the results shown in the tables above. It should be noted that Matlab gives fifteen significant digit accuracy.

Having praised its usefulness, we must also point out the major deficiency in the implementation of momentum in Matlab. Although documented in the manual [3], there seems to be no reference as to why Matlab uses the following interpretation of the weight update rule for back propagation with momentum:

\[
\Delta w_t = \alpha \Delta w_{t-1} + (1 - \alpha) \eta \text{error}
\]

It was a simple change in the code to make an implementation in line with Rumelhart’s [15] original algorithm for verifying the relevant columns of table 3.

Implementation of on-line training is difficult due to the matrix-oriented code in Matlab, whereby the presentation of patterns is considered quite irrelevant to training characteristics.

The standard Matlab [9] graph utilities are available for analysing any output or function of a neural network, or statistics across many runs, for those familiar with the basic package.

\(^1\) Matlab is a registered trademark of The MathWorks, Inc.
3.2 NevProp

*NevProp* is the simulator made freely available by the Nevada University Center for Biomedical Research, and is based loosely on Fahlman’s Quickprop software (section 3.4). It has many useful features added specifically for their research, extending the original software for ease of use. The package is still under development, with further enhancements expected.

This simulator verified all of the above results. One thing to note is that the Weight Decay term should be a negative number to achieve decay (as is the case for Fahlman’s simulator).

3.3 KRNN

The *Kernel Recurrent Neural Network* simulator was developed by Len Hamey in-house. Its ability to show many levels of accuracy has been good for giving the results in the above tables. It was written with reference to *NevProp* (section 3.2) and Fahlman’s Quickprop simulator (section 3.4), but its low-level components (such as real number interpretation), most algorithm implementations, and the interface were designed and coded independent of those packages.

Currently under development, the package will not be made publicly available in the near future.

3.4 Fahlman’s Quickprop Simulator

Although originally developed to showcase his Quickprop [4] algorithm and written entirely in LISP, Fahlman’s program was rewritten in C, and is quite capable of performing back propagation simulations. No further development is expected, however, on the software.

One major drawback with this simulator is the fact that it incorporates *don’t care* regions in calculating the output error. Fahlman does document this feature by advising that target values be 0.9 and 0.1 instead of 1 and 0. For verification purposes, a small change to the code removed this feature.

When testing the Quickprop algorithm itself, variations do exist between implementations as to the interpretation of the maximal growth factor. This software takes the attitude that, if the current and previous slopes are very close, the next weight change is the maximal growth factor multiplied by the expected weight change. Other implementations, such as *NevProp* (section 3.2) follow the Quickprop paper [4] to use the maximal growth factor after working out whether the current weight change would be too large. The difference between these interpretations shows up after thirty epochs (table 4), in the fourth or fifth decimal place.

3.5 Xerion

*Xerion* [24] is a package with a simple graphical user interface, good manual, and quite a lot of variation to the back propagation algorithm used to train a network. Although the flexibility has to be traded off for some complexity in setting the network up and analysing the results, it is a very useful piece of software that was developed over many years by researchers who constantly use it.

There is one major difference between this implementation and most others – a design decision was made to remove the factor of $\frac{1}{2}$ from the weight update rule, meaning that a learning rate half of what would normally be used is appropriate for any network learning with *Xerion*. Also, some of the documentation is incomplete, which made it difficult to try some of the algorithms (such as quickprop) without the experience of a long-term user. Due to this complication, *Xerion* could not be used to verify the results for the quickprop algorithm shown above.

The graphical interface provides some useful but simple graphical analysis tools.
3.6 Aspirin/MIGRAINES

The Aspirin language with MIGRAINES interface [8] is a large and thorough package which allows many varieties of neural network to be implemented and studied. The package was under on-going development over a period of time, and shows the results of such effort. The concept of a neural network definition language is best seen in Aspirin, where black boxes are designed to be anything from neural networks to data sources, which can be linked together to perform very complex operations easily.

However, one major drawback relating directly to the current research was that it was difficult to initialise weights to specific values. The usual initialisation is by $C$ procedure call over a collection of weights, and there are therefore a few hacks involved in giving values to specific weights. In this test, four procedures had to be written to initialise each group of weights (hidden and output layer node and bias weights).

Another point to note is that this package uses a lookup table of values for the sigmoid transfer function. This may not sound like a major problem, but the approximation involved in a table of 1024 entries shows up in the fourth to fifth decimal places for the tests performed above.

The MIGRAINES analysis package provides a good interface, with many gnu graphical analysis tools linked into the system, as well as providing the ability to link many neural networks together.

3.7 NeuDL

The Neural Network Definition Language is a valiant attempt at writing a $C$-like language for experimenting with neural networks. The documentation [14] is sparse, but the examples supplied show sufficiently the capabilities of the simple package. The author is difficult to find, so no support exists for the only public release of the software.

The software itself contains several questionable implementation decisions and bugs. Firstly, only an on-line mode of training is available, and an implementation of batch training is difficult to incorporate, even through source code modification. Weights are updated through accumulation of errors in nodes, not weights as independent entities with errors at their outputs. Biases seemed to be treated as special values associated with nodes, and not as weights from a fixed input node (of value 1), which means that the bias update rule is different to the weight update rule.

The software is cumbersomely written in $C+$+, and is not easy to change. Due to these deficiencies, it took some time to correct the mistakes and force the package to give results like those shown above (after batch training was introduced).

3.8 Condela

Condela is another neural network definition language — and it looks almost the same as NeuDL (section 3.7), with the slight implementation difference that gnu products like Flex and Bison are used to define theCondela language (which is therefore compiled code). The language contains some new structures which make it better than NeuDL, but also make it more complex, with very little documentation provided. There is a claim in the documentation that the package is used to teach neural networks.

Condela contains only on-line training capabilities, and could not verify the results shown in the tables based on batch training shown above. Biases are implemented as an extra layer consisting of one node with a fixed value. Initialisation of weights is poorly documented and difficult to implement.

Although great similarity exists between Condela and NeuDL, there is no direct connection between their developments.
3.9 Basis of AI

The author of the *Basis of AI* software claims that he "learned the neural network algorithms by programming them" [22], and is currently developing a text book on the internet. This is very good advice for the implementor; however, he does not set a good example with this piece of software.

This package suffers from a low-level restriction in its implementation. All real numbers (such as weights entered) are encoded as integers, which means that there is an approximation at the outset. A weight entered as 1.2 ended up as 1.2002 before it had even been used. This means that all results obtained from this software were skewed without even taking into account that all real number calculations would have an accumulative error. The package was unsuitable for our testing purposes.

3.10 Scientific American Demonstration Software

This small piece of software was used by van Camp to demonstrate neural networks in *Scientific American* [23]. He went on to write Xerion (section 3.5). The demonstration software is very simple and does not offer itself easily to variation from its original implementation. The user interface seemed to have a bug in it when viewing weights after several iterations (one of the weights’ values could not be shown unless it was the only weight viewed).

We could not recreate the batch results shown above due to the fact that the software did not implement batch learning (and no documentation was intended to show how to do so for such a small demonstration).

3.11 Matrix Backpropagation

The *Matrix Backpropagation* package lays claim to an efficient implementation of back propagation specifically for good performance on RISC machines with fast caches [1].

However, the software does not perform back propagation, but Vogl’s [25] variation, with varying learning rate (similar to that available with *Matlab*). The only way to achieve standard back propagation is to modify the source code. Also, scaling is automatically performed on the input patterns and weights, and there is some difficulty in implementing biases when using this software.

We could not modify the source to get results similar to any of those shown above.

3.12 PDP++

Not wanting to round up this collection of packages on a sour note, the arrival of the *PDP++* package, from the people who brought you the *PDP* software (the team of James McClelland), is a bright spot on this tour. The package, although extraordinarily large in terms of source (if you want all the insides intact), is a very professional job which has undergone a lot of development and constant use by the development team. The graphical interface is rich in buttons for modifying everything imaginable, but is very difficult to understand without a perusal of the large and in-depth manual supplied [2].

Although the floating point output is not easily configurable, the results seen confirm those given in the above tables with massaging of appropriate variables. In this respect, the style is reminiscent of *Xerion* (section 3.5), although more thorough and user-friendly. Some of the output windows (graphs, node outputs), are extremely useful for performance analysis.
4 Discussion of Package Results

The only thing that can be concluded from the application of the tests to the above packages is that if you want to do something well, then you have got to do it yourself.

Cynical as that might sound, let us qualify the statement by saying that what we wanted to do in this particular instance is test that algorithms are faithfully implemented in packages generally available. To do this, we had to find packages which claimed that they could simulate the back propagation algorithm and set up a network exactly as we wanted to (weights and configuration), then run specific tests and variations on the algorithm which were seen to be relevant. However, we have not claimed that this testing (or test suite) has been exhaustive, merely indicative of common variations on the standard algorithm.

Similarly, we did not set out to test every package publicly available. Most of the free packages used came from a list kept on the NetNews\textsuperscript{2}, and have no direct connection to the provider of that list, nor are they recommended by anyone other than the author or at most a few dedicated users (as is the case of \textit{Aspirin}, \textit{PDP++} or \textit{Xerion}). \textit{Matlab} was often used to verify the accuracy of results shown in the tables by writing simple modules to simulate algorithm variations. These modules were not a part of the test, as such, as they were developed for personal use.

5 Other Surveys

Other surveys conducted to test packages have often taken a different slant to the problem.

Ritter [13] rates available packages on the basis more of usability, assuming that they all work accurately and faithfully.

The MONK’s Problem survey by Thrun, et al [21] gives good benchmarking problems (based on artificial data sets) on which to compare algorithms. The implementations used in the tests appear to be those offered by advocates of the algorithms (or often the originators), so there is little unbiased analysis of whether an implementation is faithful to the algorithm as intended (rather than full of heuristics and associated fudge to ‘make it work’).

Zheng [27] and Tamburini and Davoli [20] give benchmark tests (or methods for producing them) for general classifier learning without direct reference to application of their tests.

Also, Prechelt’s benchmark problems [11] are intended to compare algorithms (with mostly real data), not their implementations. His methods for benchmarking are still an important set of rules to keep in mind while developing a package as well as an algorithm.

None of these investigators have provided results to compare against when offering their tests – either in terms of target learning/testing, or in expected epoch count.

6 Conclusions

There is great value in a test suite such as we have presented, whether developing a simulator for private use, or for testing someone else’s.

The effect of implementation differences has been discussed with relation to statistical behaviour (success rate, learning times) which are largely unaffected by minor implementation differences. Example behaviour (given initial conditions and parameters) may, for some learning tasks, be strongly affected by even minor implementation differences. There is an implication that initial weight configurations are not particularly useful in themselves unless the precise algorithm implementation is also preserved. However, initial weights will often be a ‘stable’ platform, leading to the same solution under minor algorithm variations as well as minor parameter variations (see also Kolen and Pollack [7]).

Although this paper’s intention had never been to show up deficiencies in some implementations, it can be concluded from the direct investigation of some packages that assumptions made about the accuracy and faithfulness of publicly available software are often baseless. This aspect

\textsuperscript{2} Newsgroup comp.ai.neural-nets, FAQ part 5.
of the report may be used by prospective users as a guide before committing to a package that does not implement the required levels of usability and precision for a desired algorithm.

We have pointed out that back propagation results are dependent upon initial configuration, parameters and implementation. Minor differences in any of these may have significant impact on results, and this should be taken into consideration when reporting them.

References


