Neural Network Implementation in SAS® Software

Proceedings of the Nineteenth Annual SAS Users Group International Conference

Revised April 21, 1994

Warren S. Sarle, SAS Institute Inc., Cary, NC, USA

Abstract

The estimation or training methods in the neural network literature are usually some simple form of gradient descent algorithm suitable for implementation in hardware using massively parallel computations. For ordinary computers that are not massively parallel, optimization algorithms such as those in several SAS procedures are usually far more efficient. This talk shows how to fit neural networks using SAS/OR®, SAS/ETS®, and SAS/STAT® software.

Introduction

As Sarle (1994) points out, many types of neural networks (NNs) are similar or identical to conventional statistical methods. However, many NN training methods converge slowly or not at all. Hence, for data analysis, it is usually preferable to use statistical software rather than conventional NN software.

For example:

- Second- and higher-order nets are linear models or generalized linear models with interaction terms. They can can be implemented directly with PROCs GENMOD, GLM, or RSREG.
- Functional-link nets are linear models or generalized linear models that include various transformations of the predictor variables. They can be implemented with PROC TRANSREG or with a DATA step and PROCs GENMOD, GLM, LOGISTIC, or REG.
- MLPs (multilayer perceptrons) are nonlinear regression or discriminant models. They can be implemented with PROCs NLIN, MODEL, NLP, or IML.
- AVQ (adaptive vector quantization) and counterpropagation are nonconvergent algorithms for cluster analysis. Better results can be obtained with PROC FASTCLUS. The IMPUTE option in FASTCLUS can be used for prediction.
- RBF (radial basis function) nets of various sorts can be implemented with DATA step programming and PROC REG. The FASTCLUS procedure and the DISTANCE macro may also be useful for computing RBF centers and bandwidths.
- Probabilistic neural nets are identical to kernel discriminant analysis, which can be implemented directly with PROC DISCRIM.
- Linear associative memories can be done directly with PROC REG.
- Hopfield nets require a little programming in IML, but a similar type of nearest-neighbor associative memory can be done with FASTCLUS, using the SEED= data set to provide prototypes and the IMPUTE option to retrieve memories.

Kangaroos

Training a NN is a form of numerical optimization, which can be likened to a kangaroo searching for the top of Mt. Everest. Everest is the *global optimum*, the highest mountain in the world, but the top of any other really tall mountain such as K2 (a good *local optimum*) would be satisfactory. On the other hand, the top of a small hill like Chapel Hill, NC, (a bad local optimum) would not be acceptable.

This analogy is framed in terms of maximization, while neural networks are usually discussed in terms of minimizing an error measure such as the least-squares criterion, but if you multiply the error measure by -1, it works out the same. So in this analogy, the higher the altitude, the smaller the error.

The compass directions represent the values of synaptic weights in the network. The north-south direction represents one weight, while the east-west direction represents another weight. Most networks have more than two weights, but representing additional weights would require a multidimensional landscape, which is difficult to visualize. Keep in mind that when you are training a network with more than two weights, everything gets more complicated.

Initial weights are usually chosen randomly, which means that the kangaroo is dropped by parachute somewhere over Asia by a pilot who has lost the map. If you know something about the scales of the inputs, you may be able to give the pilot adequate instructions to get the kangaroo to land near the Himalayas. However, if you make a really bad choice of distributions for the initial weights, the kangaroo may plummet into the Indian ocean and drown.

With Newton-type (second-order) algorithms, the Himalayas are covered with fog, and the kangaroo can only see a little way around her location. Judging from the local terrain, the kangaroo makes a guess about where the top of the mountain is, assuming that the mountain has a nice, smooth, quadratic shape. The kangaroo then tries to leap all the way to the top in one jump.

Since most mountains do not have a perfect quadratic shape, the kangaroo will rarely reach the top in one jump. Hence the kangaroo must *iterate*, i.e., jump repeatedly as previously described until she finds the top of a mountain. Unfortunately, there is no assurance that this mountain will be Everest.

In a stabilized Newton algorithm, the kangaroo has an altimeter, and if the jump takes her to a lower point, she backs up to where she was and takes a shorter jump. If ridge stabilization is used, the kangaroo also adjusts the direction of her jump to go up a steeper slope. If the algorithm isn't stabilized, the kangaroo may mistakenly jump to Shanghai and get served for dinner in a Chinese restaurant.

In steepest ascent with line search, the fog is *very* dense, and the kangaroo can only tell which direction leads up most steeply. The kangaroo hops in this direction until the terrain starts going down. Then the kangaroo looks around again for the new steepest ascent direction and iterates.

Using an ODE (ordinary differential equation) solver is similar to steepest ascent, except that the kangaroo crawls on all fives to the top of the nearest mountain, being sure to crawl in steepest direction at all times.

The following description of conjugate gradient methods was written by Tony Plate (1993):

> The environent for conjugate gradient search is just like that for steepest ascent with line search—the fog is dense and the kangaroo can only tell which direction leads up. The difference is that the kangaroo has some memory of the directions it has hopped in before, and the kangaroo assumes that the ridges are straight (i.e., the surface is quadratic). The kangaroo chooses a direction to hop in that is upwards, but that does not result in it going downwards in the previous directions it has hopped in. That is, it chooses an upwards direction, moving along which will not undo the work of previous steps. It hops upwards until the terrain starts going down again, then chooses another direction.

In standard backprop, the most common NN training method, the kangaroo is blind and has to feel around on the ground to make a guess about which way is up. If the kangaroo ever gets near the peak, she may jump back and forth across the peak without ever landing on it. If you use a decaying step size, the kangaroo gets tired and makes smaller and smaller hops, so if she ever gets near the peak she has a better chance of actually landing on it before the Himalayas erode away. In backprop with momentum, the kangaroo has poor traction and can't make sharp turns. With on-line training, there are frequent earthquakes, and mountains constantly appear and disappear. This makes it difficult for the blind kangaroo to tell whether she has ever reached the top of a mountain, and she has to take small hops to avoid falling into the gaping chasms that can open up at any moment.

Notice that in all the methods discussed so far, the kangaroo can hope at best to find the top of a mountain close to where she starts. In other words, these are *local ascent* methods. There's no guarantee that this mountain will be Everest, or even a very high mountain. Many methods exist to try to find the global optimum.

In simulated annealing, the kangaroo is drunk and hops around randomly for a long time. However, she gradually sobers up and the more sober she is, the more likely she is to hop up hill.

In a random multistart method, lots of kangaroos are parachuted into the Himalayas at random places. You hope that at least one of them will find Everest.

A genetic algorithm begins like random multistart. However, these kangaroos do not know that they are supposed to be looking for the top of a mountain. Every few years, you shoot the kangaroos at low altitudes and hope the ones that are left will be fruitful, multiply, and ascend. Current research suggests that fleas may be more effective than kangaroos in genetic algorithms, since their faster rate of reproduction more than compensates for their shorter hops.

A tunneling algorithm can be applied in combination with any local ascent method but requires divine intervention and a jet ski. The kangaroo first finds the top of any nearby mountain. Then the kangaroo calls upon her deity to flood the earth to the point that the waters just reach the top of the current mountain. She then gets on her jet ski, goes off in search of a higher mountain, and repeats the process until no higher mountains can be found.

Neural Learning and Backpropagation

To compute the arithmetic mean of a sample, you can add up the data and divide by the sample size. That's not how it's done with NNs, though.

A typical NN that needs to compute a mean starts with an initial guess for the mean, say m, often a random value. Each datum is presented to the net, one at a time, as the net slowly "learns" the mean over many iterations. Each time an input x is presented to the net, the estimate m is updated to be $m + c(x - m)$ where c is an exponential smoothing parameter. If c is small enough and if enough iterations are performed, the estimate m will eventually wander around in the vicinity of the mean but will never actually converge to the mean (Kosko 1992, 148-149). To obtain approximate convergence in practice, the parameter c can be gradually reduced over the series of iterations as in stochastic approximation (Robbins and Monro 1951).

Most NN training algorithms follow a similar scheme with:

- random initial estimates
- simple case-by-case updating formulas
- slow convergence or nonconvergence

even when direct analytical solutions are available. These slow, iterative methods are usually justified by appeals to biological plausibility or by engineering considerations for hardware implementations. Such considerations are irrelevant when using NNs for data analysis models.

Many NN training methods are said to be *gradient descent* methods. The object is to find values for the weights that minimize an error measure, usually least squares. Notice that, unlike the kangaroo, we are now trying to go down instead of up. Gradient descent is related to the well-know steepest descent algorithm, which is considered obsolete by numerical analysts.

In steepest descent, the negative gradient of the error measure is computed to identify the direction in which the error measure decreases most rapidly. A line search is conducted in that direction to determine a step size (or hop size for a kangaroo) that produces a satisfactory improvement in the error measure. This process is then iterated via the following formulas:

$$\begin{aligned}
w_j^{(i)} &= \text{the } j\text{th weight on the } i\text{th iteration} \\
g_j^{(i)} &= \text{partial derivative of the error measure} \\
& \text{with respect to } w_j^{(i)} \\
\eta^{(i)} &= \text{step size for the } i\text{th iteration} \\
\Delta w_j^{(i+1)} &= -\eta^{(i)} g_j^{(i)} \\
w_j^{(i+1)} &= w_j^{(i)} + \Delta w_j^{(i+1)}
\end{aligned}$$

Gradient descent as typically used in the NN literature differs from steepest descent in the following ways:

- A fixed *learning rate* η is used instead of a line search
- The weights are updated case-by-case
- The direction of the step is altered by a *momentum* term with control parameter α

The usual formulas for NN training are thus:

$$\begin{aligned} \Delta w_j^{(i+1)} &= -\eta g_j^{(i)} + \alpha \Delta w_j^{(i)} \\ w_j^{(i+1)} &= w_j^{(i)} + \Delta w_j^{(i+1)} \end{aligned}$$

When applied to a linear perceptron, this training method is called the *delta rule*, the *adaline rule*, the *Widrow-Hoff rule*, or the *LMS rule*. When applied to a multilayer perceptron, it is called the *generalized delta rule* or *standard backprop*.

The term *backpropagation* causes much confusion. Strictly speaking, *backpropagation* refers to the method for computing the error gradient for an MLP, a straightforward application of the chain rule of elementary calculus. By extension, *backpropagation* or *backprop* refers to a training method such as the generalized delta rule that uses backpropagation to compute the gradient. By further extension, a *backpropagation* network is an MLP trained by *backpropagation*.

The use of a fixed learning rate η causes numerous problems. If η is too small, learning is painfully slow. If η is too large, the error may increase instead of decrease, and the iterations may diverge. There is no automatic way to choose a good value for η. You must go through a tedious process of trial and error to find an acceptable value for η. A value that works well early in training may be too large or small later in training, so you may have to reset the learning rate several times during training.

In steepest descent and other optimization algorithms from the literature of numerical analysis, the error measure and its gradient are computed from the entire data set. Such algorithms can be stabilized in various ways, such as by using a line search, to ensure that the error measure decreases and to avoid divergence. The gradient computed from the entire data set also provides a way to decide when to terminate the algorithm, since the gradient is zero at an optimum.

In the NN literature, algorithms that compute the error measure and gradient from the entire data set are called *batch* or *off-line* training methods. Algorithms that compute the error and gradient and update the weights for each case are called *incremental* or *on-line* training methods. On-line methods are alleged to be more efficient and more likely to find the global optimum than off-line methods. There is little solid evidence for these claims. However, on-line methods are unstable because the error measure for the entire data set is unknown, and on-line methods provide no criterion for deciding when to terminate training—as Masters (1993, p. 174) says, "Train until you can't stand it any more."

The reason that on-line training works reasonably well in practice is that it is always used with *momentum*. The momentum term adds a fraction of the weight change applied for the previous observation to the gradient step computed for the current observation. The effect is to accumulate a weighted sum of gradient steps. If the learning rate is small and the momentum is close to one, this process approximates the gradient of the error measure for the entire data set.

Momentum is also used with batch training, but the effect is different. If the gradient points in the same general direction for several consecutive iterations, momentum increases the step size by a factor of approximately $1/(1-\alpha)$. Therefore, a momentum close to one is useful when a small learning rate is used. However, a large momentum can exacerbate divergence problems when a large learning rate is used.

The momentum is customarily set to 0.9 regardless of whether on-line or off-line training is used. However, it is often better to use larger momentum values, perhaps .99 or .999, for on-line training and smaller values, perhaps .5, for off-line training (Ripley 1993). The best momentum can be determined only by trial and error.

Considering how tedious and unreliable standard backprop is, it is difficult to understand why it is used so widely. Much of the NN research literature is devoted to attempts to speed up backprop. Some of these methods such as Quickprop (Fahlman 1988) and RPROP (Riedmiller and Braun 1993) are quite effective. There are also numerous references in the NN literature to conjugate gradient algorithms (Masters 1993), although Newton methods are almost always dismissed as requiring too much memory. Nevertheless, standard backprop retains its undeserved popularity.

It is rarely acknowledged that the choice of training method should depend on the size of the network as well as on various other factors that are more difficult to assess. For small networks (tens of weights), a Levenberg-Marquardt algorithm is usually a good choice. For a medium size network (hundreds of weights), quasi-Newton methods are generally faster. For large networks (thousands of weights), memory restrictions often dictate the use of conjugate gradient methods. If the number of weights exceeds the sample size, then Quickprop or RPROP may beat conventional optimization algorithms.

Training MLPs with NLP

MLPs can trained using PROC NLP without the headaches of standard backprop.

An MLP with one hidden layer is based on the following formulas:

$$\begin{aligned}
n_x &= \text{number of independent variables (inputs)} \\
n_h &= \text{number of hidden neurons} \\
x_i &= \text{independent variable (input)} \\
a_j &= \text{bias for hidden layer} \\
b_{ij} &= \text{weight from input to hidden layer} \\
g_j &= \text{net input to hidden layer} = a_j + \sum_{i=1}^{n_x} b_{ij} x_i \\
h_j &= \text{hidden layer values} = \text{act}_h(g_j) \\
c_k &= \text{bias for output (intercept)} \\
d_{jk} &= \text{weight from hidden layer to output} \\
q_k &= \text{net input to output layer} = c_k + \sum_{j=1}^{n_h} d_{jk} h_k \\
p_k &= \text{predicted value (output values)} = \text{act}_o(q_k) \\
y_k &= \text{dependent variable (training values)} \\
r_k &= \text{residual (error)} = y_k - p_k
\end{aligned}$$

These formulas can be coded directly in PROC NLP with a modest number of programming statements. Consider the ubiquitous NN benchmark, the XOR data, as shown in Figure 1. The target variable Y is the result of applying the logical operator *exclusive or* to the the two inputs X1 and X2. Since *exclusive or* is not a linear function, it cannot be modeled by a simple perceptron, but two hidden neurons allow a perfect fit.

To fit the XOR data, you can use PROC NLP as shown in Figure 2. The PROC NLP statement sets random initial values

```
title 'XOR Data';
data xor;
   input x1 x2 y;
cards;
0 0 0
0 1 1
1 0 1
1 1 0
run;
```

Figure 1: XOR Data

and creates output data sets containing the estimated weights and the residuals and predicted values. Arrays are declared for the data and weights according to the formulas above. A DO loop computes the value for each hidden neuron, with a nested DO loop to perform the summation. Another DO loop does the summation for the single output. Then the residual is computed, and the LSQ statement asks NLP to try to minimize the sum of squared residuals. Finally, the PARMS statement lists the parameters over which the optimization is to be performed.

The optimization takes 12 iterations to achieve essentially zero error as shown in Figure 3. The weights shown in Figure 4 are not uniquely determined since there are only four observations for estimating nine weights.

The predicted values are given by the variable P and the residuals by the variable R in the OUT= data set as shown in Figure 5.

If you want to analyze more than one data set, it is more convenient to write a macro than to repeat all the statements required for running PROC NLP every time. The appendix contains several macros including NETNLP, which fits a MLP with one hidden layer. These macros are intended as examples that you can modify and extend in various ways to meet your particular needs. They have been kept as simple as possible while still being of practical use. Various useful features such as missing value handling have been omitted to keep the macros (relatively) easy to understand and to fit in the available space. A much more elaborate macro, TNN, will be available Real Soon Now in alpha release from the author by email (saswss@unx.sas.com).

An example using the NETNLP macro is shown in Figure 7. This example uses another classic NN benchmark, the sine data. The weights are shown in Figure 6. The output is plotted with the target values in Figure 8.

The additive contributions of each hidden neuron to the output are shown in Figure 9. A logistic transformation is applied to the sum of these functions to produce the network output. Hidden neuron 3 produces the rise in the approximation to the sine curve from zero to $\pi/2$. Hidden neuron 2 generates the drop from $\pi/2$ to $3\pi/2$. Hidden neuron 1 yields the rise from $3\pi/2$ to 2π. It is also possible to get a good fit to the sine curve with only two hidden neurons. Doing so is left as an exercise for the reader.

You can test the generalization of the network by constructing a test data set covering a wider range of input than the training data and running the net on the test data using the NETRUN macro as shown in Figure 10. While the net interpolates well, its extrapolation is astonishingly bad, as shown in Figure 11. The poor extrapolation is caused by the ill-conditioning of the

```
proc nlp data=xor random=12 outest=est out=outxor;

   array x[2] x1 x2;  * input;
   array h[2];        * hidden layer;

   array a[2];        * bias for hidden layer;
   array b[2,2];      * weights from input to
                        hidden layer;
   *     c              bias for output;
   array d[2];        * weights from hidden layer
                        to output;

   *** loop over hidden neurons;
   do ih=1 to 2;

      *** net input to hidden neuron;
      sum=a[ih];
      do ix=1 to 2;
         sum=sum+x[ix]*b[ix,ih];
      end;

      *** apply logistic activation function
          to hidden units;
      h[ih]=1/(1+exp(-sum));

   end;

   *** compute output;
   sum=c;
   do ih=1 to 2;
      sum=sum+h[ih]*d[ih];
   end;

   *** apply logistic activation function
       to output;
   p=1/(1+exp(-sum));

   *** compute residual;
   r=y-p;

   *** do least squares estimation,
       minimizing sum of squared residuals;
   lsq r;

  *** list parameters to be estimated;
  parms a1-a2 b1-b4 c d1-d2;

run;

proc print data=outxor;
   var x1 x2 y p r;
run;
```

Figure 2: Fitting the XOR Data with PROC NLP

```
Iter rest nfun act   optcrit  difcrit maxgrad  lambda     rho
   1*   0    2   0    0.5305   0.1128   0.123  0.0580   0.844
   2*   0    8   0    0.1173   0.4132   0.111 0.00032   1.703
   3*   0    9   0    0.0372   0.0801  0.0478 -264E-8   1.366
   4*   0   10   0    0.0129   0.0243  0.0187       0   1.308
   5*   0   11   0  0.004578  0.00828 0.00712   -19E-8   1.288
   6*   0   12   0  0.001652  0.00293 0.00267       0   1.278
   7*   0   13   0  0.000601  0.00105 0.00099       0   1.273
   8*   0   14   0  0.000220 0.000381 0.00037       0   1.269
   9*   0   15   0 0.0000803 0.000139 0.00014 -35E-10   1.268
  10*   0   16   0 0.0000295 0.000051 0.00005 -12E-10   1.267
  11*   0   17   0 0.0000108 0.000019 0.00002       0   1.265
  12*   0   18   0 3.9753E-6 6.849E-6 6.88E-6       0   1.266
```

Figure 3: PROC NLP: Iteration History for XOR Data

```
                 Optimization Results
                  Parameter Estimates
------------------------------------------------
    Parameter           Estimate        Gradient
------------------------------------------------
   1    A1              7.944826    8.6398712E-8
   2    A2              1.213197    -0.000001622
   3    B1            -14.627206    8.9679468E-8
   4    B2             16.978973    5.838791E-13
   5    B3             19.842467    -5.59229E-15
   6    B4             -9.073972    2.2337602E-8
   7    C              23.791875    -0.000006876
   8    D1            -17.476975    -0.000002775
   9    D2            -17.571541    -0.000003699
```

Figure 4: PROC NLP: Weights for XOR Data

```
 OBS    X1    X2    Y       P            R

  1      0     0    0    0.00073    -.0007282
  2      0     1    1    0.99818    0.0018181
  3      1     0    1    0.99797    0.0020284
  4      1     1    0    0.00001    -.0000129
```

Figure 5: PROC NLP OUT= Data Set for XOR Data

```
                 Optimization Results
                 Parameter Estimates
----------------------------------------------------------
  Parameter     Estimate        Gradient    Label
----------------------------------------------------------
 1    _A1      16.095125    -0.000000478    Bias -> Hidden 1
 2    _A2       1.365670    -0.000003428    Bias -> Hidden 2
 3    _A3       2.132948     0.000004908    Bias -> Hidden 3
 4    _B1      -2.891674    -0.000002724    ANGLE -> Hidden 1
 5    _B2      -0.432451    -0.000000268    ANGLE -> Hidden 2
 6    _B3      -2.944393     0.000006864    ANGLE -> Hidden 3
 7    _C1      -3.967490    -0.000000617    Bias -> SINE
 8    _D1      -7.425998    -0.000000671    Hidden 1 -> SINE
 9    _D2      22.704353    -0.000000539    Hidden 2 -> SINE
10    _D3      -7.318340    -4.366334E-8    Hidden 3 -> SINE
```

Figure 6: PROC NLP: Weights for Sine Data

```
title 'Sine Data';
data sine;
   do angle=0 to 6.3 by .05;
      sine=sin(angle)/2+.5;
      output;
   end;
run;

%netnlp(data=sine, xvar=angle, yvar=sine,
        hidden=3, random=12);
%netrun(data=sine, xvar=angle, yvar=sine,
        out=outsine);

proc plot data=outsine;
   plot _p1*angle='-' sine*angle='*' /
        overlay;
   plot _add1*angle='1' _add2*angle='2'
        _add3*angle='3' / overlay;
run;
```

Figure 7: Sine Data

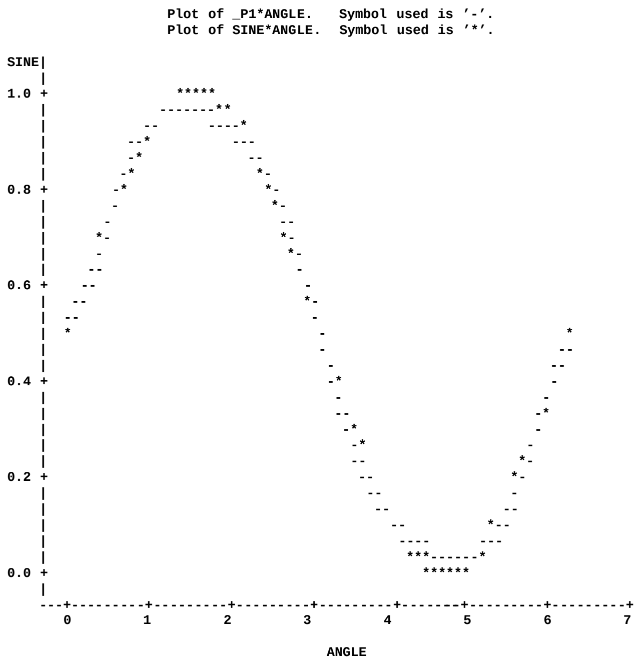

Figure 8: Plot of Sine Data: Output and Target Values

```
                  Plot of _ADD1*ANGLE.  Symbol used is ’1’.
                  Plot of _ADD2*ANGLE.  Symbol used is ’2’.
                  Plot of _ADD3*ANGLE.  Symbol used is ’3’.

 _ADD1-_ADD3
 20 +
    |
    |  22222
    |      222222
    |           222222
 15 +                22222
    |                     22222
    |                         22222
    |                             22222
    |                                 22222
 10 +                                      2222
    |                                          2222
    |                                              22222
    |                                                  222222
    |                                                       222222
  5 +                                                            222222
    |
    |
    |
    |
  0 +                   333333333333333333333333333333333333333333333333
    |              33333                                            111
    |            333                                              11
    |          33                                               11
    |        333                                              11
 -5 +      333                                              11
    |   333                                             1111
    |  111111111111111111111111111111111111111111111111111
    |
    |
-10 +
    |
    ---+---------+---------+---------+---------+---------+---------+---------+--
       0         1         2         3         4         5         6         7

                                        ANGLE
```

Figure 9: Plot of Additive Components of Output for Sine Data

```
data test;
   do angle=-6.3 to 12.5 by .1;
      sine=sin(angle)/2+.5;
      output;
   end;
run;

%netrun(data=test, xvar=angle, yvar=sine,
        out=outtest);

proc plot data=outtest;
   plot _p1*angle=’-’ sine*angle=’*’ /
        overlay;
   plot _add1*angle=’1’ _add2*angle=’2’
        _add3*angle=’3’ / overlay;
run;
```

Figure 10: Test Data

network, a common problem in MLPs.

In the NN literature, ill-conditioning is dealt with by weight decay and regularization, which are extensions to the nonlinear case of ridge regression and other forms of shrinkage estimators that are used in linear regression. With PROC NLP it is easier to impose an upper bound on the norm of the weights, which accomplishes virtually the same thing as weight decay but in a more directly interpretable way. In Figure 12, an upper bound of 10 is placed on the Euclidean norm of the weight vector. The results are plotted for the test data in Figure 13, which shows much better extrapolation, although the range of accurate extrapolation is still limited.

A common problem with MLPs is local minima. Specifying a different seed for the pseudorandom number generator can lead to a local minimum as shown Figure 14, where the seed was changed from 12 to 123. The output is plotted in Figure 15. The simplest way to deal with local optima is to try many different sets of random initial weights. You can do this by writing a macro with a %DO loop that runs the NETNLP macro many times for a few iterations each and then chooses the best result among these to initialize the final optimization. The TNN macro has options for doing this.

```
                   Plot of _P1*ANGLE.   Symbol used is ’-’.
                   Plot of SINE*ANGLE.  Symbol used is ’*’.

_P1 |
    |
1.0 +       **                   **                  ***
    |  ---------                ---*                 * **
    |      *  *----            -- --                **  *
    |     *   *    --          -   -                *   *
    |     *    *     -         *    -               *   *
    |          *      -       -     -              *    *
0.8 +    *     *       -      -     *-             *     *
    |    *      *      --     *      *                   *
    |    *      *       -     -      -             *
    |   *               --   *       -            *      *
    |           *        -   -       *            *      *
    |   *        *       --  -        -
0.6 +   *                 -  -        -           *        *
    |            *        ---         *          *        *
    |  *         *         --         -
    |  *                    *                   *         *
    |             *         *          -        --         *         *
    |             *                    *       - --                 *
0.4 +                      *           -       *   -       *
    |             *        *           *       -    -      *        *
    |              *                   -*      -    -       *       *
    |              *       *            -      *    --
    |                     *             *     -     -       *      *
    |              *      *             -     *      -      *      *
0.2 +               *                   -*    -      --      *     *
    |               *     *              -    -       -      *    *
    |               *    *               -   -         --         *
    |                *   *                -  -          --   **   *
    |                *  **               -- --            ----*  *
    |                ** *                *---                ---------
0.0 +                 **                  **                   **
    |
    ---+---------+---------+---------+---------+---------+---------+---------+-
     -6.3      -3.3      -0.3       2.7       5.7       8.7      11.7      14.7

                                       ANGLE
```

Figure 11: Plot of Test Data

```
%netnlp(data=sine, xvar=angle, yvar=sine,
         hidden=3, constrn=10, random=12);
   %netrun(data=test, xvar=angle, yvar=sine,
         out=outtest);

   proc plot data=outtest;
      plot _p1*angle='-' sine*angle='*' /
         overlay;
      plot _add1*angle='1' _add2*angle='2'
         _add3*angle='3' / overlay;
   run;
```

Figure 12: Sine Data with Bounds on the Weights

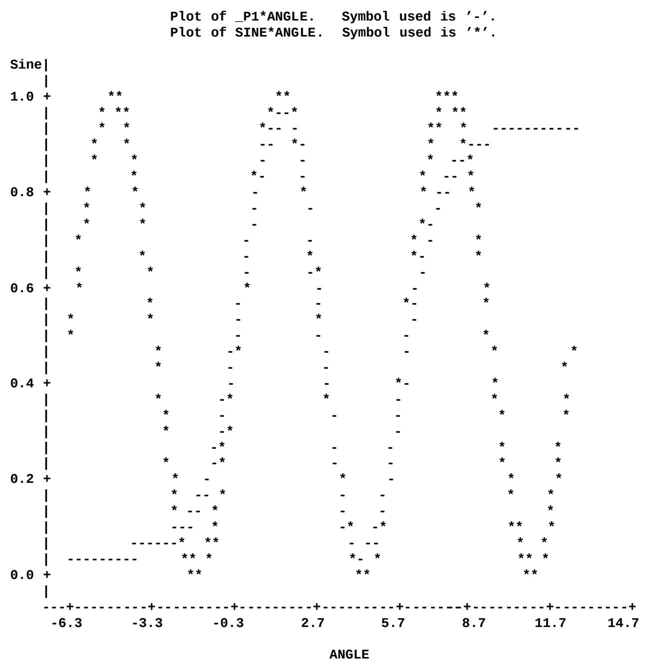

Figure 13: Plot of Test Data with Bounded Weights

```
%netnlp(data=sine, xvar=angle, yvar=sine,
         hidden=3, random=123);
   %netrun(data=sine, xvar=angle, yvar=sine,
         out=outlocal);

   proc plot data=outlocal;
      plot _p1*angle='-' sine*angle='*' /
         overlay;
   run;
```

Figure 14: Local Optimum with the Sine Data

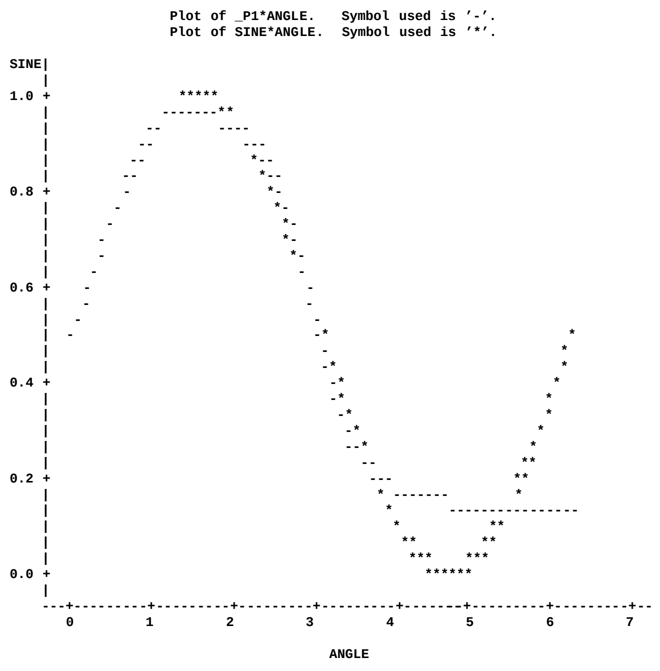

Figure 15: Plot Showing Local Optimum for the Sine Data

Forecasting with PROC MODEL

Extrapolation does not work well with NNs or other flexible models. However, if you treat the sine data as a time series, you can forecast it accurately with a NN. Forecasting is easier with the MODEL procedure than with the NLP procedure because PROC MODEL has a variety of features designed for forecasting, including processing of lagged variables, which are called *shift registers* or *tapped delay lines* in the NN literature.

PROC MODEL does not handle arrays very well, so the code for the MLP computations must be written with macro loops as in the NETMODEL macro in the appendix. The macro loops make the code unpleasant to look at because of all the %s and &s, but the result is more efficient than the code used in NETNLP. However, the optimization algorithms in PROC MODEL are less efficient than those in PROC NLP, especially for models with many parameters.

Figure 16 shows a DATA step that generates a longer segment of the sine curve to allow for lag processing and also includes further ANGLE values with missing values for the SINE variable to show forecasts. The forecasts are plotted in Figure 17.

```
data sinem;
   do angle=0 to 20 by .1;
      if angle<10
         then sine=sin(angle)/2+.5;
         else sine=.;
      output;
   end;
run;

%netmodel(data=sinem, yvar=sine,
   xvar=lag5(sine) lag10(sine),
   fitrange=angle to 10,
   hidden=2, random=12, outfore=outfore);

proc format;
   value $sym
      'ACTUAL'='*'
      'PREDICT'='-';
run;

proc plot data=outfore;
   format _type_ $sym1.;
   plot sine*angle=_type_;
   where _type_ in ('ACTUAL' 'PREDICT');
run;
```

Figure 16: Forecasting the Sine Data

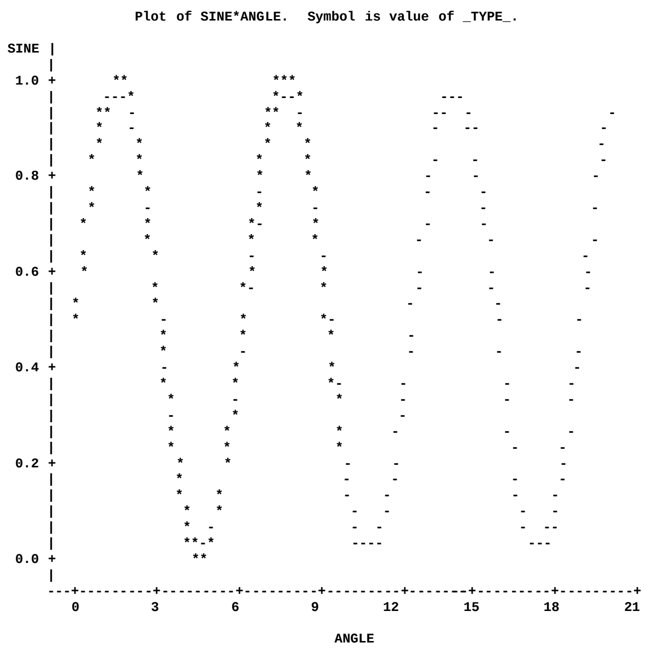

Figure 17: Plot Showing Forecasts from PROC MODEL

Counterpropagation and FASTCLUS

Counterpropagation clusters the training cases and computes predicted values as the means of the nearest cluster to each case. You can obtain similar results more efficiently with the FASTCLUS procedure using the IMPUTE option for predictions. The CPROP1 and CPROP2 macros in the appendix perform the FASTCLUS equivalent of unidirectional and bidirectional counterpropagation, respectively, and the CPRUN macro makes the predictions for either case. The complications relate mainly to standardizing the data. The predicted values for the sine data with 9 clusters are shown in Figure 18.

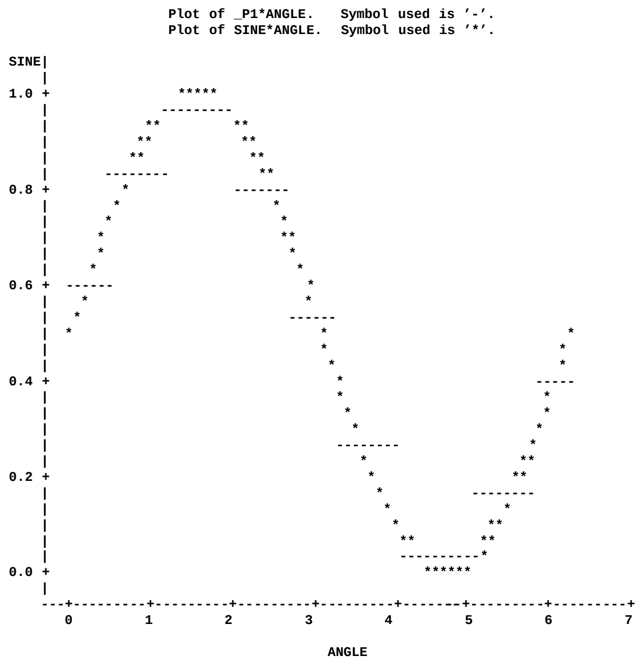

Figure 18: Plot of CPROP Output for the Sine Data

RBF Networks, FASTCLUS, and REG

There are many variations on RBF networks. The RBF macro in the appendix uses FASTCLUS to find the RBF centers and bandwidths. Then a DATA step computes the RBF values, which are used as regressors in a linear regression with PROC REG. The predicted values for the sine data with 7 clusters are shown in Figure 19.

The Motorcycle Data

The data from Schmidt, Mattern, and Schüler (1981), used often in examples of nonparametric regression, are interesting because of the strong nonlinearity and marked heteroscedasticity. The input values are time in milliseconds after a motorcycle impact. The target values are acceleration of the head of the test object in *g* (acceleration due to gravity).

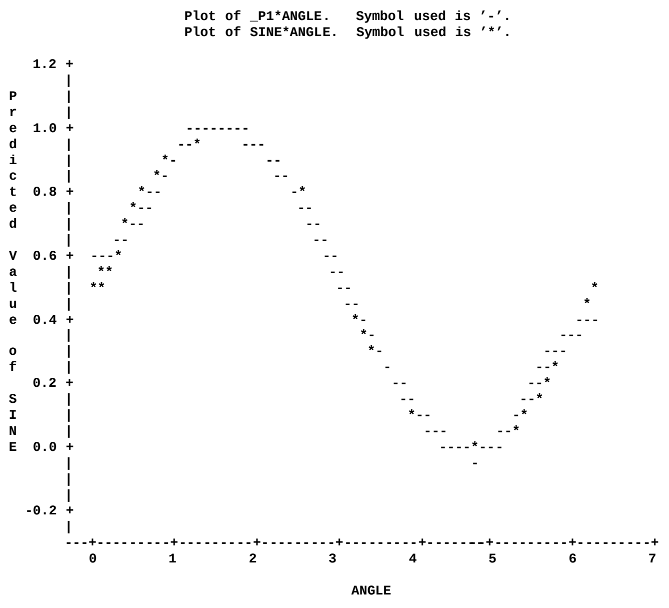

Figure 19: Plot of RBF Output for the Sine Data

The outputs from MLPs with one through five hidden neurons are shown in Figures 20 through 24. One hidden neuron is clearly inadequate, but the fit is good with two hidden neurons and excellent with four or more.

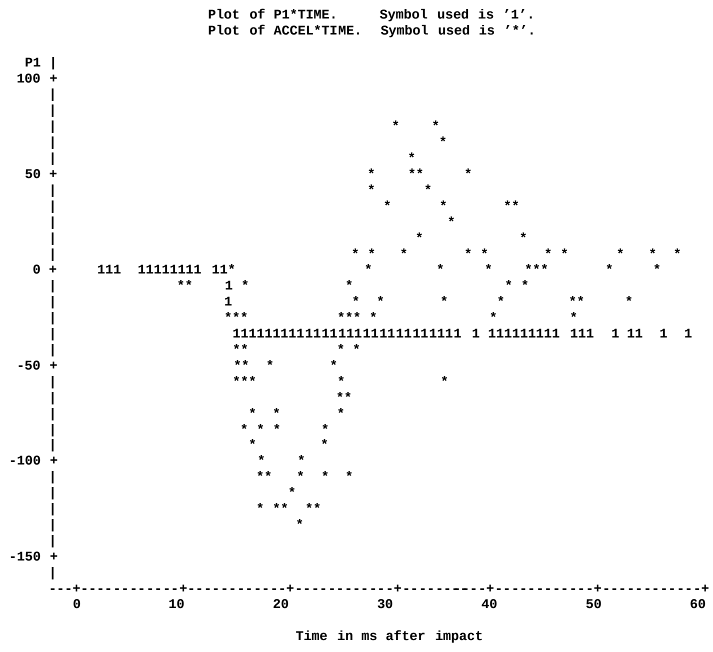

Figure 20: MLP Output with 1 Hidden Neuron for the Motorcycle Data

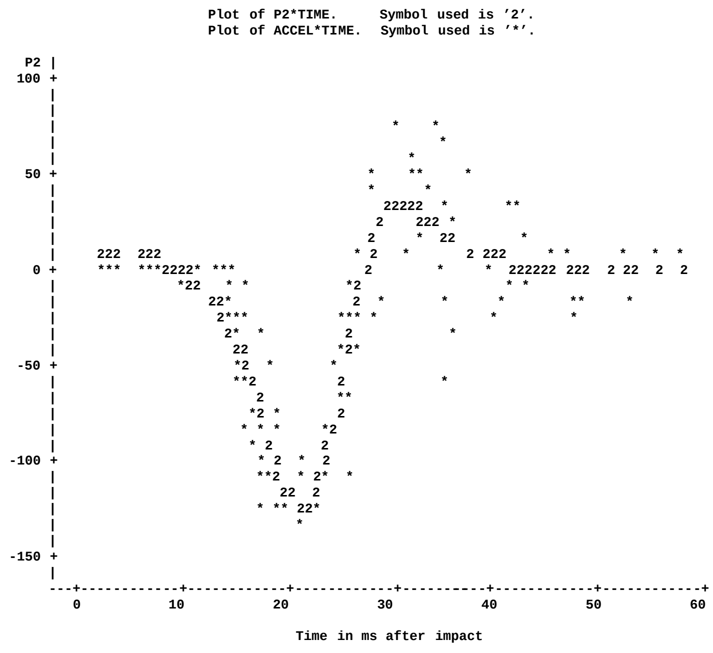

Figure 21: MLP Output with 2 Hidden Neurons for the Motorcycle Data

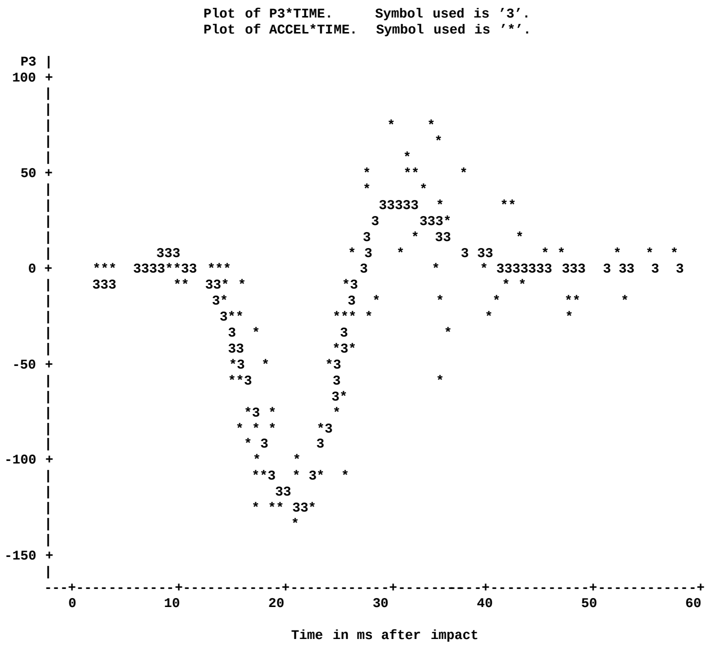

Figure 22: MLP Output with 3 Hidden Neurons for the Motorcycle Data

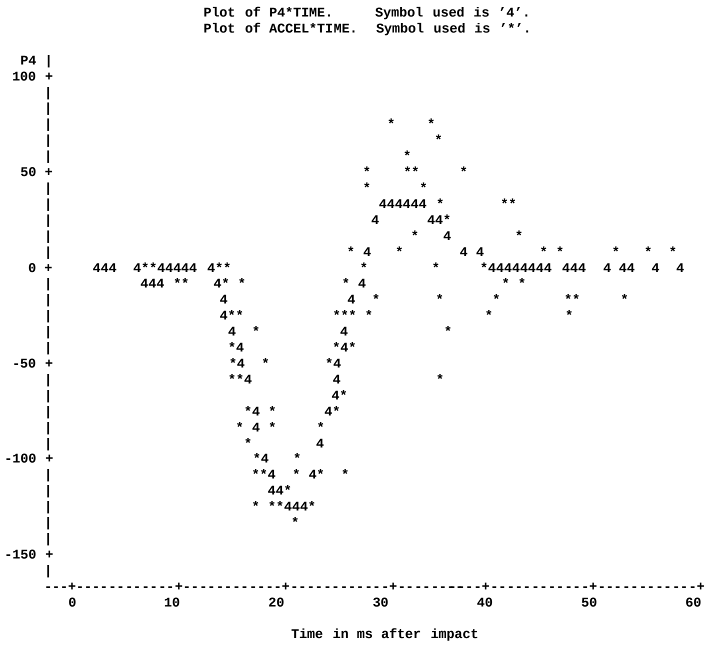

Figure 23: MLP Output with 4 Hidden Neurons for the Motorcycle Data

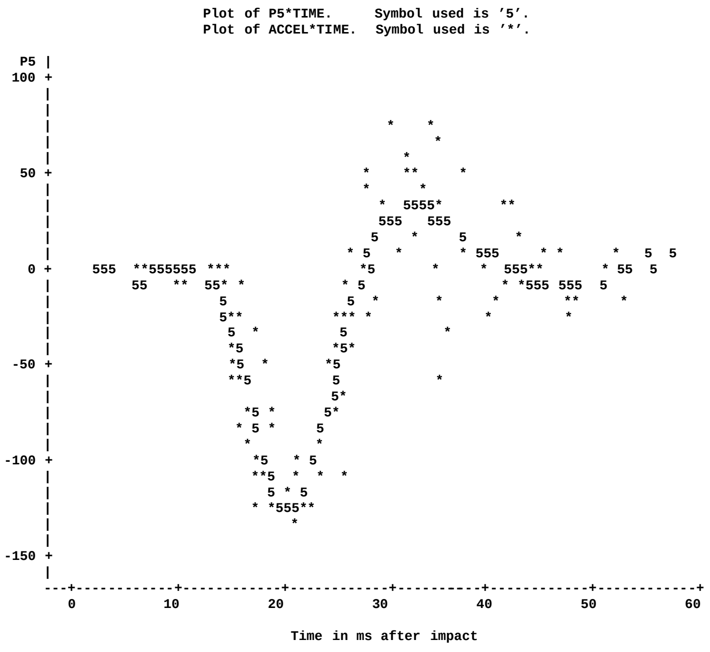

Figure 24: MLP Output with 5 Hidden Neurons for the Motorcycle Data

Figures 25 through 28 show the output of CPROP1 and CPROP2 with 10 and 20 clusters.

Figures 29 through 33 show the output of RBF with 4, 8, 12, 16, and 20 clusters. Twelve clusters give the best fit by eye, but the right tail is too rough.

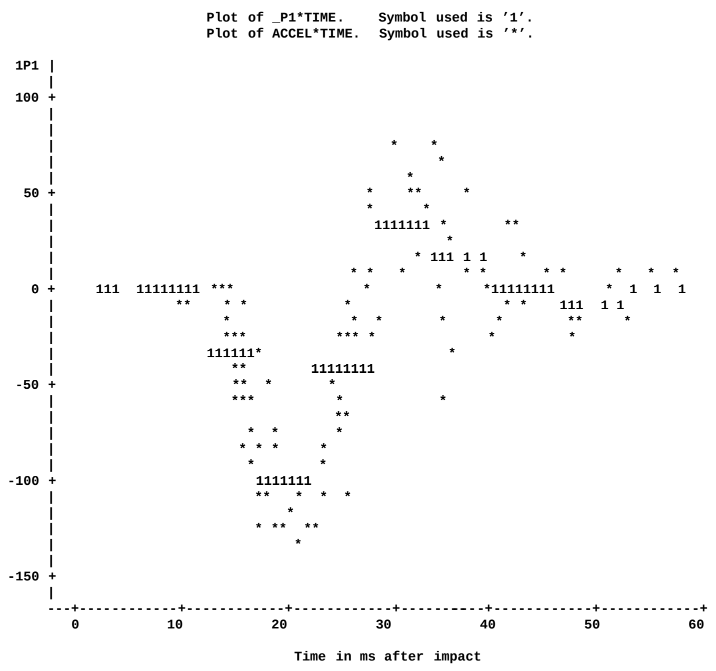

Figure 25: CPROP1 Output with 10 Hidden Neurons for the Motorcycle Data

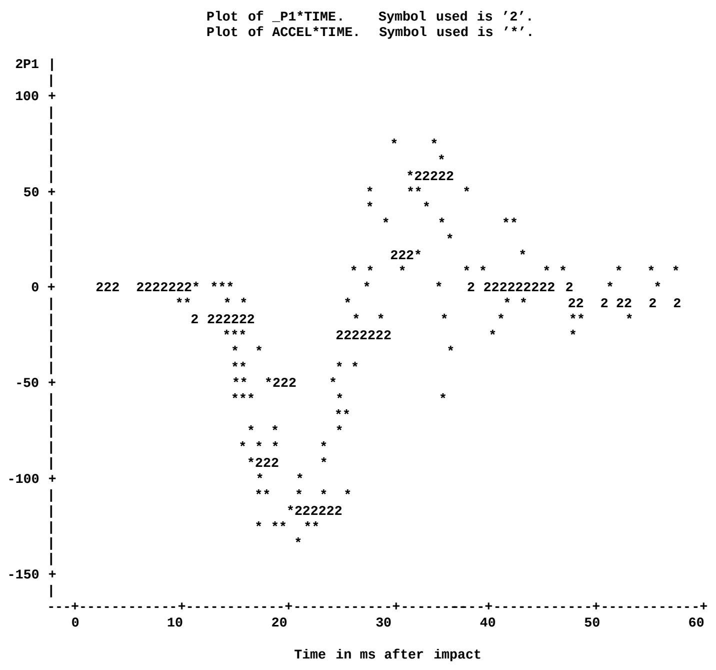

Figure 26: CPROP2 Output with 10 Hidden Neurons for the Motorcycle Data

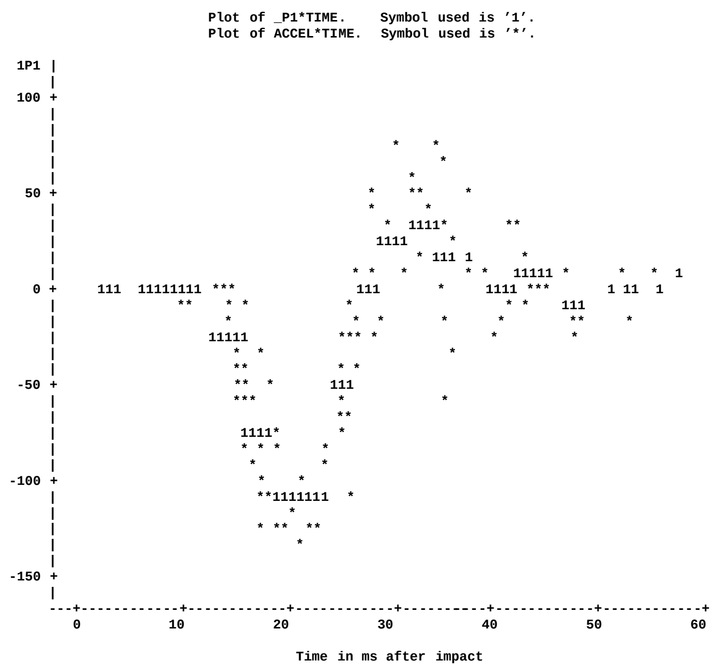

Figure 27: CPROP1 Output with 20 Hidden Neurons for the Motorcycle Data

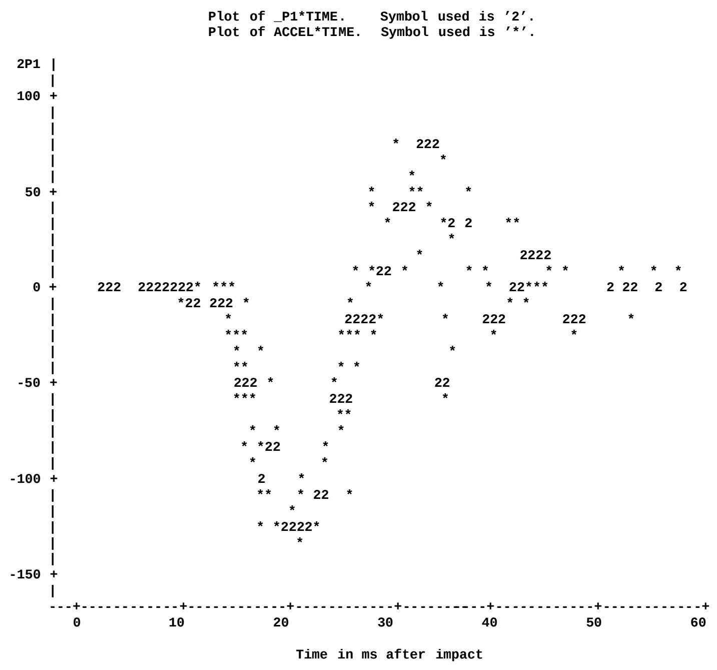

Figure 28: CPROP2 Output with 20 Hidden Neurons for the Motorcycle Data

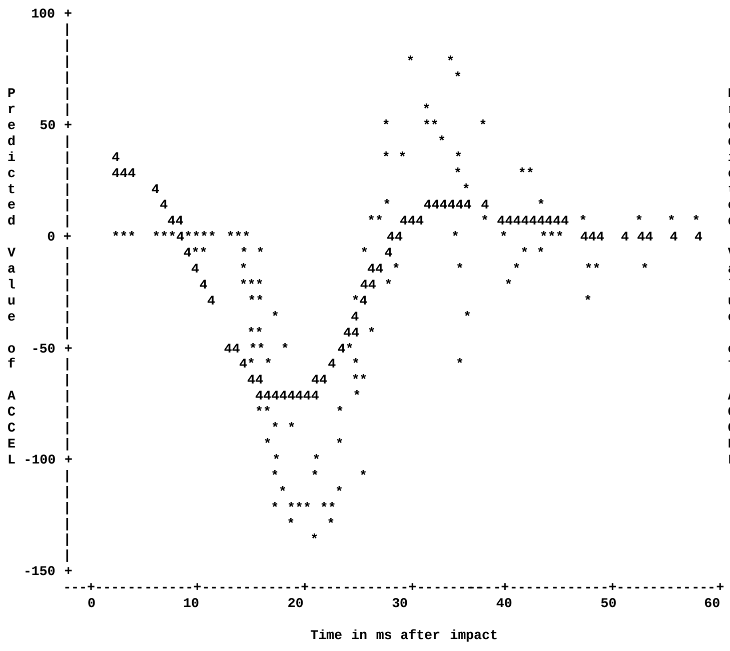

Figure 29: RBF Output with 4 Hidden Neurons for the Motorcycle Data

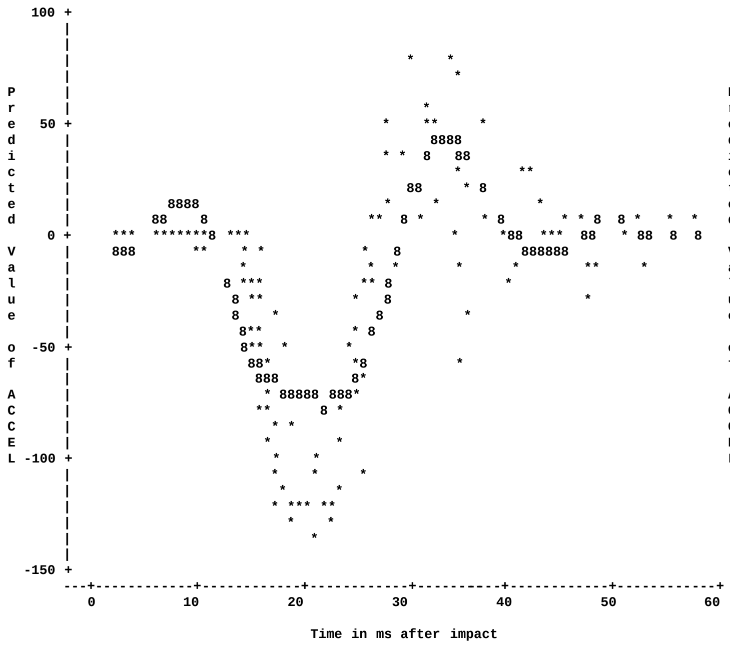

Figure 30: RBF Output with 8 Hidden Neurons for the Motorcycle Data

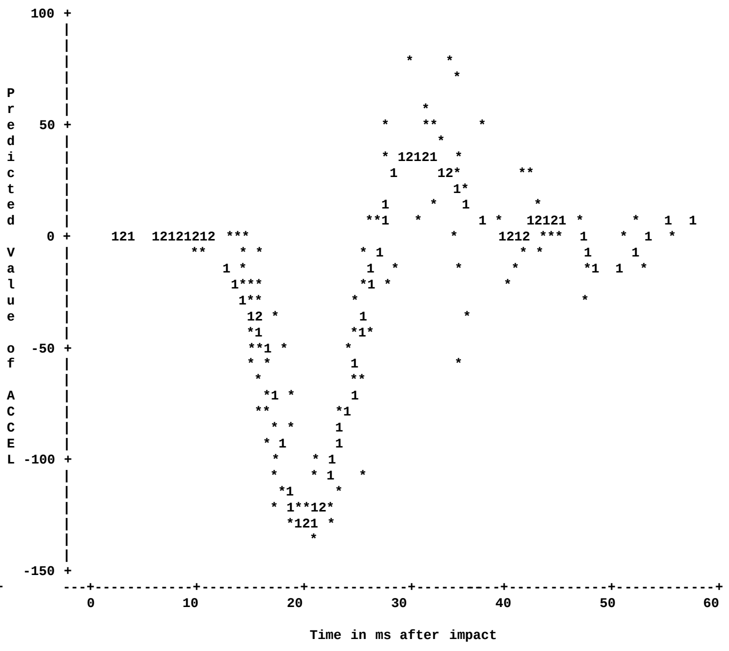

Figure 31: RBF Output with 12 Hidden Neurons for the Motorcycle Data

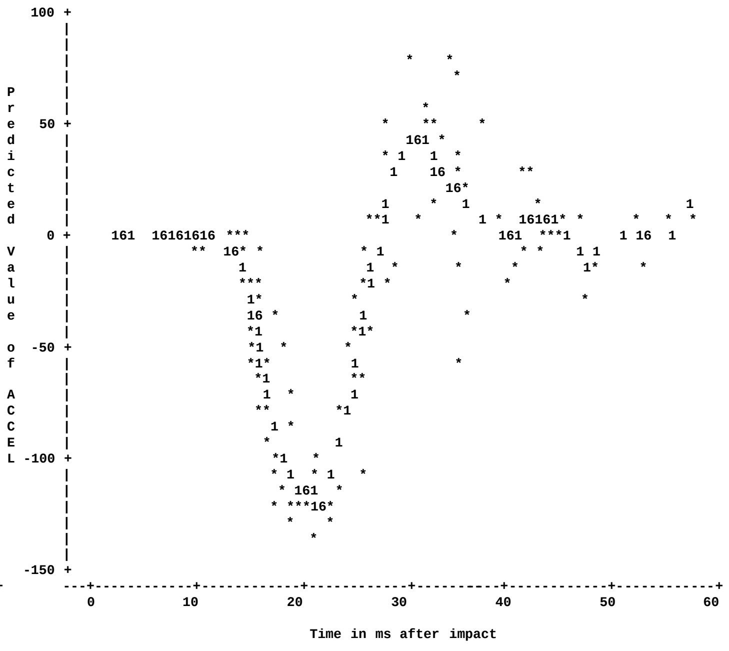

Figure 32: RBF Output with 16 Hidden Neurons for the Motorcycle Data

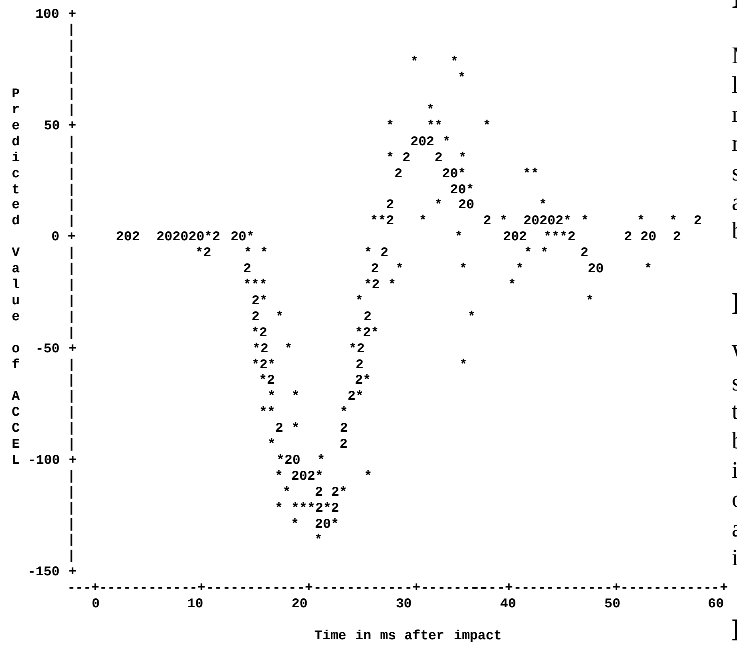

Figure 33: RBF Output with 20 Hidden Neurons for the Motorcycle Data

The table below shows the root mean squared error (RMSE) and root final prediction error (RFPE) for each network fitted to the motorcycle data. The degrees of freedom are taken to be the number of parameters, which is correct for the CPROP1 and RBF networks and a reasonable approximation for the other networks. RMSE is an estimate of the prediction error using the unknown optimal weights in the population. RFPE is an estimate of the prediction error using the weights estimated from the sample to make predictions for the population. Based on RFPE, the best models would be a MLP with two or three hidden neurons or a RBF network with 16 hidden neurons.

Architecture	Hidden Neurons	Model DF	RMSE	RFPE
MLP	1	4	47.59	48.30
	2	7	23.24	23.84
	3	10	22.87	23.72
	4	13	22.92	24.02
	5	16	23.62	25.00
CPROP1	10	10	30.67	31.80
	20	20	25.04	26.86
CPROP2	10	10	34.12	35.38
	20	20	31.56	33.85
RBF	4	4	33.71	34.21
	8	8	30.86	31.77
	12	12	23.68	24.72
	16	16	22.62	23.94
	20	20	22.83	24.49

Neural Network Myths

Many myths and half-truths have been promulgated in the NN literature. Articles in the popular press are often uncritical and make exaggerated claims that most responsible NN researchers readily disavow. Other claims are perpetuated as folklore, often stemming from speculations based on little evidence. Some are based on confusion between different types of networks, or between different parts of a network.

Neural networks are intelligent

We have seen that NN learning is nothing more than a form of statistical estimation, often using algorithms that are slower than the algorithms used in statistical software. It is true that there have been some impressive results using NNs for applications such as image processing. The intelligence in these applications is that of the researchers who have found effective input representations and designed the network architectures. There is no intelligence inherent in the networks.

Neural networks generalize

Some authors overgeneralize about the ability of NNs to generalize. According to Medsker, Turban, and Trippi (1993), "When a neural network is presented with noisy, incomplete, or previously unseen input, it generates a reasonable response." The ability of NNs to generalize is similar to that of other statistical models. It is vital to distinguish between interpolation and extrapolation. If a network of suitable complexity is properly trained on data that adequately cover the range of the inputs, and if the function to be approximated is sufficiently smooth, then it is possible for a NN to interpolate well. Extrapolation, however, is much more dangerous with a flexible nonlinear model such as an MLP than it is with linear regression; even in the linear case, extrapolation is risky. Figure 11 demonstrates how bad NN extrapolation can be.

Neural networks are fault tolerant

Nelson and Illingworth (1991, 64) repeat the cliche that, "Neural networks are extremely fault tolerant and degrade gracefully." Fault tolerance is the ability to produce approximately correct output when some of the neurons malfunction. Some types of networks are indeed fault tolerant. Networks based on a large number of neurons with local effects are inherently fault tolerant. Kernel regression, a conventional statistical method, is an excellent example of such a fault tolerant method. On the other hand, MLPs are often ill-conditioned In an ill-conditioned net, just as in an ill-conditioned linear model, tiny errors can easily blow up into wild outputs. In such cases it is essential to use careful regularization when training the net.

Neural networks are superior to statistical methods

Since many NN models are similar or identical to statistical models, the falsity of this claim is obvious to anyone familiar with statistics. However, Hecht-Nielsen (1990, 121) says, "In essence, in terms of its everyday practice, there has been only modest progress in regression analysis since the days of Gauss." The only adequate counter-argument to such a sweeping pronouncement is a texbook on regression such as Weisberg (1985) or Myers (1986).

Neural networks are faster than statistical algorithms because of parallel processing

Most practical applications of NNs for data analysis use serial computers, so the advantages of parallel processing are inapplicable. Furthermore, many efficient parallel algorithms have been developed for use with conventional statistical methods. Both NNs and conventional statistical methods can benefit from parallel computers.

Local optima are rare

Rumelhart, Hinton, and Williams (1986, 332) say that, "We do not know the frequency of such local minima, but our experience with this [XOR] and other problems is that they are quite rare." The generalized delta rule is suprisingly resistant to local optima in the XOR problem, possibly because the XOR problem was used extensively in developing the generalized delta rule. Nevertheless, local optima are easily demonstrable in the XOR data (Kolen and Pollack 1991), the sine data (see Figure 15), and many other other common NN benchmarks.

Inputs need to be normalized to lie between 0 and 1

This myth is trivial but amazingly widespread. The fact that activation functions usually have an output range of 0 to 1 has apparently mislead some people into thinking that the inputs should also be between 0 and 1. Nelson and Illingworth (1991, 177) say, "The next step requires scaling the data to be in the range 0 to 1 ... so that we may use the sigmoid transfer function. Because the sigmoid function does not recognize negative numbers, we have used absolute values and will rely on the network to work out the correct relationships." Actually, the logistic function accepts any real-valued input. There are some cases in which it is useful for all the weights in a network to be of the same order of magnitude, such as for weight decay, and there are some algorithms that may work better if the inputs and targets are normalized in some way, such as standard backprop, but there are no strict requirements on the range of the inputs.

References

Fahlman, S.E. (1988) "An empirical study of learning speed in back-propagation networks," CMU-CS-88-162, School of Computer Science, Carnegie Mellon University.

Hecht-Nielsen, R. (1990), *Neurocomputing*, Reading, MA: Addison-Wesley.

Kolen, J.F and Pollack, J.B. (1991), "Back Propagation is Sensitive to Initial Conditions," in Lippmann, R.P., Moody, J.E., and Touretzky, D.S., eds., *Advances in Neural Information Processing Systems 3*, 860-867, San Mateo, CA: Morgan-Kaufmann.

Kosko, B. (1992), *Neural Networks and Fuzzy Systems*, Englewood Cliffs, NJ: Prentice Hall.

Masters, T. (1993), *Practical Neural Network Recipes in C++*, New York: Academic Press.

Medsker, L., Turban, E., and Trippi, R.R. (1993), "Neural Network Fundamentals for Financial Analysts," in Trippi, R.R. and Turban, E., eds., *Neural Networks in Finance and Investing*, 3-25, Chicago: Probus.

Myers, R.H. (1986), *Classical and Modern Regression with Applications*, Boston: Duxbury Press

Nelson, M.C. and Illingworth, W.T. (1991), *A Practical Guide to Neural Nets*, Reading, MA: Addison-Wesley.

Plate, T. (1993), "Re: Kangaroos (Was Re: BackProp without Calculus?)," Usenet article <93Sep8.162519edt.997@neuron.ai.toronto.edu> in comp.ai.neural-nets.

Riedmiller, M. and Braun, H. (1993), "A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm," *Proceedings of the IEEE International Conference on Neural Networks* 1993, San Francisco: IEEE.

Ripley, B.D. (1993), "Statistical Aspects of Neural Networks," in Barndorff-Nielsen, O.E., Jensen, J.L. and Kendall, W.S., eds., *Networks and Chaos: Statistical and Probabilistic Aspects*, London: Chapman & Hall.

Robbins, H. and Monro, S. (1951), "A Stochastic Approximation Method," *Annals of Mathematical Statistics*, 22, 400-407.

Sarle, W.S. (1994), "Neural Networks and Statistical Models," *Proceedings of the Nineteenth Annual SAS Users Group International Conference*, Cary, NC: SAS Institute.

Schmidt, G., Mattern, R., and Schüler, F. (1981), "Biomechanical Investigation to Determine Physical and Traumatological Differentiation Criteria for the Maximum Load Capacity of Head and Vertebral Column With and Without Protective Helmet Under Effects of Impact," EEC Research Program on Biomechanics of Impacts, Final Report Phase III, Project 65, Institut für Rechtsmedizin, Universität Heidelberg, Heidelberg, Germany.

Weisberg, S. (1985), *Applied Linear Regression*, New York: John Wiley & Sons.

Appendix: Macros

Macro	Product Required	Purpose
NETNLP	SAS/OR	Train a multilayer perceptron with one hidden layer by least-squares using PROC NLP
NETRUN	---	Run a multilayer perceptron with a DATA step
NETMODEL	SAS/ETS	Train and run a multilayer perceptron with one hidden layer by least-squares using PROC MODEL
CPROP1	SAS/STAT	Train a unidirectional counterprop network
CPROP2	SAS/STAT	Train a bidirectional counterprop network
CPRUN	SAS/STAT	Run a counterprop network
RBF	SAS/STAT	Train a radial basis function network
RBFRUN	---	Run a radial basis function network

These macros run under release 6.07.03 or later. They have not been tested in earlier releases. NETNLP will not run in releases prior to 6.07.03 because of the absence of PROC NLP. You should use the latest release available to you because of performance enhancements in the NLP and MODEL procedures. NETNLP and NETMODEL may produce different results in different releases for singular models or models with multiple local optima.

```
************************** NETNLP Macro ****************************;
%macro netnlp(    /* Train a multilayer perceptron with PROC NLP */
   data=DATA???,  /* Data set containing inputs and training values */
   xvar=XVAR???,  /* List of variables that are inputs */
   yvar=YVAR???,  /* List of variables that have training values */
   hidden=2,      /* Number of hidden neurons */
   inest=,        /* Data set containing initial weight estimates.
                     If not specified, random initial estimates are
                     used. */
   random=0,      /* Seed for random numbers for initial weights */
   maxiter=1000,  /* Maximum number of training iterations allowed */
   constrn=,      /* optional constraint on norm of weights (requires
                     6.10 or later release) */
   outest=_EST_,  /* Data set to contain estimated weights */
   nlpopt=SHORT PINIT HISTORY);
                  /* Additional options to be passed to PROC NLP,
                     such as TECH= or ABSCONV=. Do NOT put commas
                     between the NLP options! */

   ****** number of variables;
   %nxy

   ****** number of hidden neurons;
   %global nh;
   %let nh=&hidden;

   ****** number of weights;
   %nweight
   %if &nparm=0 %then %goto exit;

   ****** random initial values;
   %netinit
   ****** training;
   proc nlp start=0 maxfunc=32000 maxiter=&maxiter
      data=&data inest=&inest outest=&outest &nlpopt;

      *** run the net;
      %network(NLP)

      *** define the objective function;
      lsq _r1-_r&ny;

      *** define the weights;
      parms &parms;

      *** nonlinear constraints;
      %constrn

      *** labels for the weights;
      %netlabel

   run;
%exit:
%mend netnlp;


************************** NETRUN Macro ****************************;
%macro netrun(    /* Run a multilayer perceptron with a DATA step */
   data=DATA???,  /* Data set containing inputs and training values */
   xvar=XVAR???,  /* List of variables that are inputs */
   yvar=YVAR???,  /* List of variables that have training values */
   hidden=,       /* Number of hidden neurons; defaults to value from
                     previous execution of one of the NET... macros */
   test=0,        /* Specify TEST=1 if these are test data, not
                     training data */
   inest=_EST_,   /* Data set containing weights */
   out=_DATA_);   /* Data set to contain outputs and residuals */

   ****** number of variables;
   %nxy

   ****** number of hidden neurons;
   %global nh;
   %if %bquote(&hidden)^= %then %let nh=&hidden;

   ****** number of weights;
   %nweight
   %if &nparm=0 %then %goto exit;

   ****** run the network;
   data &out(keep=&xvar &yvar _h1-_h&nh _p1-_p&ny _r1-_r&ny
                 _add1-_add&nh);

      *** read estimated weights;
      if _n_=1 then set &inest(where=(_type_='PARMS'));

      *** read input and training values;
      set &data end=_end;

      *** compute outputs and residuals;
      %network(DATA)
      *** compute SSE and additive components;
      %netstat

      *** print statistics;
      %pstat(MLP)

   run;
%exit:
%mend netrun;


*************************** NETMODEL Macro ****************************;
%macro netmodel( /*Train & run a multilayer perceptron with PROC MODEL*/
   data=DATA???,  /* Data set containing inputs and training values */
   xvar=XVAR???,  /* List of variables that are inputs.
                     Do not use abbreviated lists with -, -- or :. */
   yvar=YVAR???,  /* List of variables that have training values
                     Do not use abbreviated lists with -, -- or :. */
   weight=,       /* Variable that contains observation weights */
   id=,           /* ID variable to identify observations */
   outvars=,      /* Additional variables to be included in output
                     data sets */
   fitrange=,     /* RANGE statement specification to be used for
                     fitting the model; solving uses the entire data
                     set */
   hidden=2,      /* Number of hidden nodes */
   inest=,        /* Data set containing initial weight estimates.
                     If not specified, random initial estimates are
                     used. */
   random=0,      /* Seed for random numbers for initial weights */
   outest=_EST_,  /* Data set containing estimated weights */
   out=,          /* Output data set to contain observed and predicted
                     values and residuals in separate observations
                     identified by the _TYPE_ variable */
   outfore=,      /* Output data set to contain forecasts */
   nahead=,       /* Value of n for n-step-ahead forecasts */
   maxiter=1000,  /* Maximum number of training iterations allowed;
                     MAXITER=-1 suppresses training so you can use
                     NETMODEL for computing outputs or forecasts only */
   modelopt=,     /* Additional options to be passed to PROC MODEL
                     in the PROC MODEL statement */
   fitopt=METHOD=MARQUARDT ITPRINT OUTALL OUTCOV SINGULAR=1E-8,
                  /* Additional options to be passed to PROC MODEL
                     in the FIT statement */
   solveopt=NOPRINT SINGLE OUTALL);
                  /* Additional options to be passed to PROC MODEL
                     in the SOLVE statement */

   %if %index(&xvar &yvar,-:) %then %do;
      %put %qcmpres(ERROR: Abbreviated variable lists are not allowed
           by the NETMODEL macro.);
      %goto exit;
   %end;
   %local i ix ih iy;

   ****** number of variables;
   %nxy

   ****** number of hidden neurons;
   %global nh;
   %let nh=&hidden;

   ****** number of weights;
   %nweight
   %if &nparm=0 %then %goto exit;

   ****** random initial values;
   %netinit

   ****** training;
   proc model data=&data &modelopt;

      %if %bquote(&id)^= %then id &id %str(;);
      %if %bquote(&outvars)^= %then outvars &outvars %str(;);
      %if %bquote(&weight)^= %then weight &weight %str(;);
      %if %bquote(&fitrange)^= %then range %unquote(&fitrange) %str(;);

      *** labels for weights;
      %netlabel

      *** run the net;
      %network(MODEL)

      *** fit the model;
      %if &maxiter>=0 %then %do;
         fit &yvar / estdata=&inest outest=&outest maxiter=&maxiter
              %if %bquote(&out)^= %then out=&out;
              &fitopt;
         run;
      %end;


      *** make forecasts;
      %if %bquote(&outfore)^= %then %do;
         %if %bquote(&fitrange)^= %then range %scan(&fitrange,1) %str(;);
         solve &yvar / estdata=&outest out=&outfore
            %if %bquote(&nahead)^= %then nahead=&nahead;
            &solveopt;
         run;
      %end;

%exit:
%mend netmodel;


************************** CPROP1 Macro *****************************;
%macro cprop1(     /* Train a unidirectional counterprop network */
   data=DATA???,  /* Data set containing inputs and training values */
   xvar=XVAR???,  /* List of variables that are inputs */
   yvar=YVAR???,  /* List of variables that have training values */
   maxc=MAXC???,  /* Maximum number of clusters */
   maxiter=10,    /* Maximum number of iterations */
   inest=,       /* Data set containing initial cluster seeds.
                    If not specified, the usual FASTCLUS seed
                    selection algorithm is used. */
   outest=_EST_); /* Data set to contain cluster centers */

   ****** number of variables;
   %nxy
   ****** standardize the inputs;
   proc means noprint data=&data;
      var &xvar;
      output out=_std_;
   run;
   %standard(&data,_std_,&xvar,&nx)

   ****** cluster the inputs;
   proc fastclus noprint data=_temp_ outseed=_seed_ out=_clus_
      maxc=&maxc maxiter=&maxiter
      %if %bquote(&inest)^= %then seed=&inest;
      ;
      var &xvar;
   run;

   ****** find means of the targets;
   proc means noprint nway data=_clus_;
      var &yvar;
      output out=_ymean_ mean=&yvar;
      class cluster;
   run;

   ****** save estimates in one data set;
   data &outest;
      retain &yvar 1;
      if _end=0 then do;
         set _std_ end=_end;
         output;
      end;
      if _end=1 then do;
         _stat_=’ ’;
         merge _seed_ _ymean_; by cluster;
         output;
      end;
   run;

%mend cprop1;


************************** CPROP2 Macro ****************************;
%macro cprop2(    /* Train a bidirectional counterprop network */
   data=DATA???,  /* Data set containing inputs and training values */
   xvar=XVAR???,  /* One set of variables */
   yvar=YVAR???,  /* The other set of variables */
   maxc=MAXC???,  /* Maximum number of clusters */
   maxiter=10,    /* Maximum number of iterations */
   ratio=1,       /* Relative weight of Y variables to X variables */
   inest=,        /* Data set containing initial cluster seeds.
                     If not specified, the usual FASTCLUS seed
                     selection algorithm is used. */
   outest=_EST_); /* Data set to contain cluster centers */

   ****** number of variables;
   %nxy
   %let nv=%eval(&nx+&ny);

   ****** standardize variables;
   proc means noprint data=&data;
      var &xvar &yvar;
      output out=_std_;
   run;
   %if %bquote(&ratio)^=1 %then %do;
      data _std_; set _std_;
         array _y &yvar;
         if _stat_='STD' then do over _y; _y=_y/&ratio; end;
      run;
   %end;
   %standard(&data,_std_,&xvar &yvar,&nv)

   ****** cluster using both sets of variables;
   proc fastclus noprint data=_temp_ outseed=_seed_ maxc=&maxc maxiter=&maxiter
      %if %bquote(&inest)^= %then seed=&inest;
      ;
      var &xvar &yvar;
   run;

   ****** save estimates in one data set;
   data &outest; set _std_ _seed_; run;

%mend cprop2;


************************** CPRUN Macro ******************************;
%macro cprun(      /* Run a counterprop network */
   data=DATA???,   /* Data set containing inputs and training values */
   xvar=XVAR???,   /* List of variables that are inputs */
   yvar=YVAR???,   /* List of variables that have training values */
   test=0,         /* Specify TEST=1 if these are test data, not
                      training data */
   inest=_EST_,    /* Data set containing cluster centers */
   out=_DATA_);    /* Data set to contain outputs and residuals */

   ****** number of variables;
   %nxy
   %let nv=%eval(&nx+&ny);

   ****** standardize variables;
   %standard(&data,&inest,&xvar,&nx,_p1-_p&ny)

   ****** figure out number of clusters;
   data _null_;
      if 0 then set &inest nobs=_n;
      call symput('maxc',trim(left(put(_n,12.))));
      stop;
   run;

   ****** impute target values;
   proc fastclus noprint data=_temp_ out=_out_ maxc=&maxc impute maxiter=0
      seed=&inest(where=(_stat_=' ')
                  rename=(%do i=1 %to &ny; &&y&i=_p&i %end;))
      ;
      var &xvar _p1-_p&ny;
   run;

   ****** unstandardize variables and compute statistics;
   %let nh=%eval(&maxc-5);
   %let nparm=%eval(&nh*&ny);
   data &out;
      retain _m1-_m&nv _s1-_s&nv .;
      if _n_=1 then do;
```

```
         set &inest(where=(_stat_='MEAN'));
         array _v &xvar &yvar;
         array _m _m1-_m&nv;
         array _s _s1-_s&nv;
         do over _v; _m=_v; end;
         set &inest(where=(_stat_='STD'));
         do over _v; _s=_v; end;
      end;
      set _out_ end=_end;
      array _u &xvar _p1-_p&ny;
      do over _u; _u=_u*_s+_m; end;
      array _y &yvar;
      array _p _p1-_p&ny;
      array _r _r1-_r&ny;
      do over _y; _r=_y-_p; _sse_+_r**2; end;
      _n=n(of _r1-_r&ny);
      _nobs_+_n;
      %pstat(CProp)
   run;

%mend cprun;


************************** RBF Macro ******************************;
%macro rbf(       /* Train a radial basis function network */
   data=DATA???,  /* Data set containing inputs and training values */
   xvar=XVAR???,  /* List of variables that are inputs */
   yvar=YVAR???,  /* List of variables that have training values */
   maxc=MAXC???,  /* Maximum number of clusters */
   maxiter=10,    /* Maximum number of iterations */
   smooth=2,      /* Smoothing parameter */
   inest=,        /* Data set containing initial cluster seeds.
                     If not specified, the usual FASTCLUS seed
                     selection algorithm is used. */
   out=,          /* Data set to contain outputs and residuals */
   outest=_EST_); /* Data set to contain cluster centers and weights */

   ****** number of variables;
   %nxy

   ****** standardize the inputs;
   proc means noprint data=&data;
      var &xvar;
      output out=_std_;
   run;
   %standard(&data,_std_,&xvar,&nx)

   ****** cluster the inputs;
   proc fastclus noprint data=_temp_ outseed=_seed_ out=_cls_
      maxc=&maxc maxiter=&maxiter
      %if %bquote(&inest)^= %then seed=&inest;
      ;
      var &xvar;
   run;

   ****** figure out the number of clusters;
   data _null_;
      if 0 then set _seed_ nobs=_n;
      call symput('nclus',trim(left(put(_n,12.))));
      stop;
   run;

   data _tmpest_;
      set _std_ _seed_;
      retain _nclus_ &nclus _smooth_ &smooth;
      drop _freq_ _crit_ _radius_ _near_ _gap_;
   run;

   ****** compute radial basis functions;
   %rbfunc(0)

   ****** compute weights for RBFs;
   proc reg data=_rbf_ outest=_reg_;
      model &yvar=_rbf1-_rbf&nclus / noint
         selection=rsquare start=&nclus stop=&nclus rmse jp sbc;
      %if %bquote(&out)^= %then %do;
         output out=&out p=_p1-_p&ny r=_r1-_r&ny;
      %end;
   run;

   ****** save estimates in one data set;
   proc transpose data=_reg_ out=_tmpreg_;
      var _rbf:;
      id _depvar_;
   run;
   data _tmpreg_; set _tmpreg_;
      cluster=_n_;
      keep cluster &yvar;
   run;
   data &outest;
      merge _tmpest_ _tmpreg_;
      by cluster;
   run;

%mend rbf;

************************** RBFRUN Macro *****************************;
%macro rbfrun(    /* Run a radial basis function network */
   data=DATA???,  /* Data set containing inputs and training values */
   xvar=XVAR???,  /* List of variables that are inputs */
   yvar=YVAR???,  /* List of variables that have training values */
   test=0,        /* Specify TEST=1 if these are test data, not
                     training data */
   inest=_EST_,   /* Data set containing cluster centers and weights */
   out=_DATA_);   /* Data set to contain outputs and residuals */

   ****** number of variables;
   %nxy

   ****** figure out the number of clusters and smoothing parameter;
   data _null_; set &inest;
      call symput('nclus',trim(left(put(_nclus_,12.))));
      call symput('smooth',trim(left(put(_smooth_,12.))));
      stop;
   run;

   ****** compute radial basis functions and outputs;
   %rbfunc(1)

%mend rbfrun;
```

```
************************* Utility Macros *************************;
****** The macros below are not called directly by the user ******;

************ NXY Macro ************;
%macro nxy; %* count variables and store names in macro arrays
   %global nx ny;
   %if %index(&xvar &yvar,%str(%()) %then %do;
      %local i token;
      %do i=1 %to 999999;
         %let token=%qscan(&xvar,&i,%str( ));
         %if &token= %then %goto xbreak;
         %global x&i;
         %let x&i=%unquote(&token);
      %end;
%xbreak:
      %let nx=%eval(&i-1);
      %do i=1 %to 999999;
         %let token=%qscan(&yvar,&i,%str( ));
         %if &token= %then %goto ybreak;
         %global y&i;
         %let y&i=%unquote(&token);
      %end;
%ybreak:
      %let ny=%eval(&i-1);
   %end;
   %else %do;
      data _null_;
         if 0 then set &data;
         array _x[*] &xvar;
         array _y[*] &yvar;
         call symput("nx",
            trim(left(put(dim(_x),12.))));
         call symput("ny",
            trim(left(put(dim(_y),12.))));
         length _mname _vname $8;
         do _i=1 to dim(_x);
            _mname='x'||trim(left(put(_i,12.)));
            call execute('%global '||_mname);
            call vname(_x[_i],_vname);
            call symput(_mname,trim(_vname));
         end;
         do _i=1 to dim(_y);
            _mname='y'||trim(left(put(_i,12.)));
            call execute('%global '||_mname);
            call vname(_y[_i],_vname);
            call symput(_mname,trim(_vname));
         end;
         stop;
      run;
   %end;
%mend nxy;

************ NWEIGHT Macro ************;
%macro nweight; %* number of weights (parameters) for MLPs;
   %global na nb nc nd nparm parms;
   %****** size of each parameter array;
   %let na=&nh;
   %let nb=%eval(&nx*&nh);
   %let nc=&ny;
   %let nd=%eval(&nh*&ny);
   %****** number of parameters;
   %let nparm=%eval(&na+&nb+&nc+&nd);
   %if &nh>10000 | &nparm>30000 | &nparm<0 %then %do;
      %put %qcmpres(ERROR: Too many weights (&nparm) or
         hidden neurons (&nh).);
      %let nparm=0;
   %end;
   %****** list of all parameters;
   %else %let parms=_a1-_a&na _b1-_b&nb
                    _c1-_c&nc _d1-_d&nd;
%mend nweight;

************ NETINIT Macro ************;
%macro netinit;  %* random initial values for MLPs;
   %if %bquote(&inest)= %then %do;
      %let inest=_INEST_;
      data &inest(type=est);
         _type_='PARMS';
         retain &parms 0;
         array _ab _a: _b:;
         do over _ab;
            _ab=ranuni(&random)-.5;
         end;
      run;
   %end;
%mend netinit;

************ NETWORK Macro ************;
%macro network(proc); %* code for 3-layer perceptron usingmacros;
   *** compute hidden layer;
   %let i=0;
   %do ih=1 %to &nh;
      _sum=_a&ih
      %do ix=1 %to &nx;
         %let i=%eval(&i+1);
         +&&x&ix*_b&i
      %end;
      ;
      * apply activation function to hidden node;
      %acthid(_h&ih,_sum);
   %end;
   *** compute output;
   %let i=0;
   %do iy=1 %to &ny;
      _sum=_c&iy
      %do ih=1 %to &nh;
         %let i=%eval(&i+1);
         +_h&ih*_d&i
      %end;
      ;
      * apply activation function to output;
      %if &proc=MODEL %then %do;
         %actout(&&y&iy,_sum)
      %end;
      %else %do;
         %actout(_p&iy,_sum)
         _r&iy=&&y&iy-_p&iy;
      %end;
   %end;
%mend network;

************ CONSTRN Macro ************;
%macro constrn; %* constrain the Euclidean norm of the weights;
   %if %bquote(&constrn) ne %then %do;
      array _parm[*] &parms;
      wnorm=0;
      do _i=1 to &nparm;
         wnorm+_parm[_i]**2;
      end;
      wnorm=sqrt(wnorm/&nparm);
      nlincon wnorm < &constrn;
   %end;
%mend constrn;

************ NETLABEL Macro ************;
%macro netlabel; %* labels for the weights in MLPs;
   label
   %let i=0;
   %do ih=1 %to &nh;
      _a&ih="Bias -> Hidden &ih"
      %do ix=1 %to &nx;
         %let i=%eval(&i+1);
         _b&i="&&x&ix -> Hidden &ih"
      %end;
   %end;
   %let i=0;
   %do iy=1 %to &ny;
      _c&iy="Bias -> &&y&iy"
      %do ih=1 %to &hidden;
         %let i=%eval(&i+1);
         _d&i="Hidden &ih -> &&y&iy"
      %end;
   %end;
   ;
%mend netlabel;

************ STANDARD Macro ************;
%macro standard(data,stat,var,nv,retain); %* standardize variables;
   data _temp_;
      retain &retain _m1-_m&nv _s1-_s&nv .;
      if _n_=1 then do;
         set &stat(where=(_stat_='MEAN'));
         array _v[&nv] &var;
         array _m[&nv] _m1-_m&nv;
         array _s[&nv] _s1-_s&nv;
         drop _m1-_m&nv _s1-_s&nv
              _type_ _freq_ _stat_ _i;
         do _i=1 to &nv; _m[_i]=_v[_i]; end;
         set &stat(where=(_stat_='STD'));
         do _i=1 to &nv;
            if _v[_i] then _s[_i]=_v[_i];
                      else _s[_i]=1;
         end;
      end;
      set &data;
      do _i=1 to &nv; _v[_i]=(_v[_i]-_m[_i])/_s[_i]; end;
   run;
%mend standard;
```

```
************ RBFUNC Macro ************;
%macro rbfunc(ypr);
   %if &ypr %then %do;
      data &out;
         set &inest(in=_est keep=&xvar &yvar
                    _stat_ _rmsstd_)
             &data end=_end;
   %end;
   %else %do;;
      data _rbf_;
         set _tmpest_(in=_est keep=&xvar
                      _stat_ _rmsstd_) &data;
   %end;
      array _x [&nx] &xvar;
      array _m [&nx] _temporary_;
      array _s [&nx] _temporary_;
      array _c [&nclus,&nx] _temporary_;
      %if &ypr %then %do;
         array _y [&ny] &yvar;
         array _p [&ny];
         array _r [&ny];
         array _b [&nclus,&nx] _temporary_;
      %end;
      array _h [&nclus] _temporary_;
      array _rbf [&nclus] _rbf1-_rbf&nclus;
      retain _first 1 _m: _s: _c: _h:;
      drop _first _nc _i _d _stat_ _rmsstd_;
      if _est then do;
         if _stat_='MEAN' then do _i=1 to &nx;
            _m[_i]=_x[_i]; end; else
         if _stat_='STD'  then do _i=1 to &nx;
            _s[_i]=_x[_i]; end; else
         if _stat_=' '    then do;
            _nc+1;
            _h[_nc]=_rmsstd_;
            do _i=1 to &nx;
               _c[_nc,_i]=_x[_i]*_s[_i]+_m[_i];
            end;
            %if &ypr %then %do;
               do _i=1 to &ny;
                  _b[_nc,_i]=_y[_i];
               end;
            %end;
         end;
      end;
      else do;
         if _first then do;
            _first=0;
            %if &nx=1 %then %do;
               _d=_s[1];
            %end;
            %else %do;
               _d=sqrt(uss(of _s[*]));
            %end;
            do _nc=1 to &nclus;
               if _h[_nc]>0 then if _h[_nc]<_d
                  then _d=_h[_nc];
            end;
            do _nc=1 to &nclus;
```

```
               if _h[_nc]<=0 then _h[_nc]=_d;
               _h[_nc]=(_h[_nc]*&smooth)**2*&nx*2;
            end;
         end;
         do _nc=1 to &nclus;
            _d=0;
            do _i=1 to &nx;
               _d+((_x[_i]-_c[_nc,_i])/_s[_i])**2;
            end;
            _rbf[_nc]=exp(-_d/_h[_nc])/_h[_nc];
         end;
         _d=sum(of _rbf[*]);
         do _nc=1 to &nclus;
            _rbf[_nc]=_rbf[_nc]/_d;
         end;
         %if &ypr %then %do;
            do _i=1 to &ny;
               _d=0;
               do _nc=1 to &nclus;
                  _d+_rbf[_nc]*_b[_nc,_i];
               end;
               _p[_i]=_d;
               _r[_i]=_y[_i]-_p[_i];
               _sse_+_r[_i]**2;
               _nobs_+1;
            end;
         %end;
         output;
      end;
      %if &ypr %then %do;
         %let nh=&nclus;
         %let nparm=&nclus;
         %pstat(RBF)
      %end;
   run;
%mend rbfunc;

************ NETSTAT Macro ************;
%macro netstat; %* compute fit statistics for the net;
      *** compute additive components for graphing;
      %let i=0;
      %do iy=1 %to &ny;
         %do ih=1 %to &nh;
            %let i=%eval(&i+1);
            _add&i=_h&ih*_d&i;
         %end;
      %end;
      *** compute fit statistics;
      _n=n(of _r1-_r&ny);
      _nobs_+_n;
      _sse_
      %do iy=1 %to &ny;
         +_r&iy**2
      %end;
      ;
%mend netstat;

************ PSTAT Macro ************;
%macro pstat(net); %* print fit statistics for the net;
   drop _np_ _nobs_ _df_ _sbc_ _rfpe_ _mse_ _rmse_ _sse_;
   if _end then do;
      _np_=&nparm;
      %if &test %then %do;
         _df_=_nobs_;
         _sbc_=.;
         _rfpe_=.;
      %end;
      %else %do;
         _df_=max(0,_nobs_-_np_);
         _sbc_=_nobs_*log(_sse_/_nobs_)+
               _np_*log(_nobs_);
         _rfpe_=sqrt(_sse_*(_nobs_+_np_)/(_nobs_*_df_));
      %end;
      _mse_=_sse_/_df_;
      _rmse_=sqrt(_mse_);

      file print;
      put "Network. . . . . . . . . . . . &net"
        / "Data . . . . . . . . . . . . . &data"
        / "Input Variables. . . . . . . . &xvar"
        / "Output Variables . . . . . . . &yvar"
        / "Number of Hidden Neurons . . . &nh"
        / "Number of Parameters . . . . . "  _np_
      %if &test %then %do;
        / "Number of Test Data Values . . "  _nobs_
      %end;
      %else %do;
        / "Number of Training Values. . . "  _nobs_
      %end;
        / "Degrees of Freedom for Error . "  _df_
        / "Sum of Squared Errors. . . . . "  _sse_
        / "Mean Squared Error . . . . . . "  _mse_
        / "Root Mean Squared Error. . . . "  _rmse_
      %if &test=0 %then %do;
        / "Root Final Prediction Error. . "  _rfpe_
        / "Schwarz’s Bayesian Criterion . "  _sbc_
      %end;
        ;
   end;
%mend pstat;

************ Activation Function Macros *********;
%macro acthid(out,in);    %* activation function for hidden layer;
   if &in<-45 then &out=0;    * avoid overflow;
   else if &in>45 then &out=1;*    or underflow;
   else &out=1/(1+exp(-&in)); * logistic function;
%mend acthid;

%macro actout(out,in);    %* activation function for output;
   %acthid(&out,&in)
%mend actout;

%*
```