Neural Network Tutorial:
Use of train.c and nnTrain.m

Chuck Anderson
Department of Computer Science
Colorado State University
Fort Collins, CO 80523
http://www.cs.colostate.edu/~anderson

August 7, 1996

1 Introduction

This tutorial briefly introduces the error back-propagation algorithm and cross-validation with early-stopping. Then the use of train.c and the Matlab function nnTrain.m are explained. This was written for students at Colorado State University who wish to use my code for class or research, but is available to all. It assumes you are familiar with Unix and C. If you have Matlab installed on your Unix system, you can use the nnTrain.m function to run train.c. The LaTeX source for this document is also available on-line to give you an example to learn from.

Please let me know if you have success or problems with this code. Also, I’d like to hear any suggestions you have for improvements. You may send me email at anderson@cs.colostate.edu or visit my web page at http://www.cs.colostate.edu/~anderson.

2 Data Sets and the Algorithm

One of the simplest and most useful neural network algorithms is the error back-propagation training algorithm applied to a feedforward network, which is a network with no cycles in its connection structure, i.e., it is not a recurrent network. Implementing this kind of network is very straightforward. I have done this in a single C source file named train.c. This tutorial explains how to run train.c independently or from within Matlab using the function named nnTrain. See the Mathworks, Inc. home page at http://www.mathworks.com for information on Matlab.

First, we must address the method used in presenting data to the network for training and for testing. train.c employs cross-validation and early stopping. The available data is divided into the following three disjoint sets:

Training set This data is used to train the network.

Validation set The error of the network averaged over this data is used to decide when the training algorithm has found the best approximation to the data without overfitting.

Testing set The best network, given by the validation test, is applied to the test set and the error averaged over the test set is taken as a prediction of how well the network will generalize to novel data.

Training is accomplished by calculating the derivative of the network’s error with respect to each weight in the network when presented a particular input pattern. This derivative indicates which direction each weight should be adjusted to reduce the error. Each weight is modified by taking a small step in this direction. With a nonzero momentum factor, a fraction of the previous weight change is added to the new weight value. This accelerates learning in some cases. The patterns in the training set are stepped through one by one. A pass through all training patterns is called an epoch. The training data is repetitively presented for multiple epochs, until a specified number of epochs have been taken.

After each epoch, the error of the network applied to the validation set of patterns is calculated. If the current network scores the lowest error so far on the validation set, this network’s weights are saved. At the conclusion of training, the network’s best weights are used to calculate the network’s error on the testing set.
3 Using train.c

After compiling train.c into an executable named train, typing the command train produces the following usage statement:

Usage: train spec-file <hammer>

Example of a spec-file:
-ninputs 1 -noutputs 1 (these must appear before file names)
-nhiddens1 5 -nhiddens2 5 (-nhiddens2 n is optional)
-train one.data two.data (list of one or more files to compose train set)
-validate three.data (optional, list of one or more for validate set)
-test four.data (list of one or more files for test set)
-orate 0.001 -hrate 0.1 (for hammer, hrate is at most 15*orate)
-mom .9 -epochs 1000
-summarize (optional, to specify short output, one line per run)
-end (says end of run specification. Now do it.)
-orate 0.01 (additional runs, all unspecified values same as previous run)
-end
-orate 0.1 -end (any whitespace may separate tokens)

Each part of the specification file (spec-file) will be explained in the order they appear in the example.

1. -ninputs: Number of input components in each pattern.
2. -noutputs: Number of output components in the patterns.
3. -nhiddens1: Number of hidden units to be used in the first, and possibly only, hidden-unit layer of the network. Start with 1, then try higher numbers. If more than one number appears here, then multiple runs with different numbers of hidden units will be performed.
4. -nhiddens2: Number of hidden units to be used in the second hidden-unit layer of the network. Can be zero, and if this line is not included, the default value is zero. If more than one number appears here then multiple runs with different numbers of hidden units will be performed.
5. -train: Names of data files to be concatenated to form training data.
6. -validate: Names of validation data files.
7. -test: Names of testing data files.
8. -orate: Learning rates, possibly more than one, to try for output layer. Start with something like 0.01 or 0.1
9. -hrate: Learning rates, possibly more than one, for hidden layer. Try values 1 to 10 times the output layer rate.
10. -mom: The momentum rates. Should be 0 or greater and less than 1.
11. -epochs: The number of passes to make through the training data set.
12. -summarize: Produce short format output. Without this token, output is in long format.
13. -end: Terminates the specification for one call to train.c. Another may start immediately after this in the spec-file. Each set of specifications must end with -end.
For an example, let’s try to approximate a sine wave with a neural network. The network will receive a single number as input, call it \( x \). The network will produce a single output that we will try to make match \( \sin(13x) \). I chose 13 for the following reason. My \( x \) values range from 0 to 1 and I wanted a value of \( x = 1 \) to roughly correspond to \( 4\pi \), to get two periods of data.) I used Matlab to produce the data. You could write a short C program to do it. Here is my Matlab code:

\[
x = [1:10] * 0.1;
sineTrain = [x ; 0.5+0.5*\sin(13*x)]';
sineValidate = [x+0.03 ; 0.5+0.5*\sin(13*(x+0.03))]';
sineTest = [x-0.03 ; 0.5+0.5*\sin(13*(x-0.03))]';
\]

This is what the matrix sineTrain contains. The first column is the \( x \) value and the second column is the corresponding target value.

<table>
<thead>
<tr>
<th>( x )</th>
<th>( \sin(13x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1000</td>
<td>0.9818</td>
</tr>
<tr>
<td>0.2000</td>
<td>0.7578</td>
</tr>
<tr>
<td>0.3000</td>
<td>0.1561</td>
</tr>
<tr>
<td>0.4000</td>
<td>0.0583</td>
</tr>
<tr>
<td>0.5000</td>
<td>0.6076</td>
</tr>
<tr>
<td>0.6000</td>
<td>0.9993</td>
</tr>
<tr>
<td>0.7000</td>
<td>0.6595</td>
</tr>
<tr>
<td>0.8000</td>
<td>0.0861</td>
</tr>
<tr>
<td>0.9000</td>
<td>0.1190</td>
</tr>
<tr>
<td>1.0000</td>
<td>0.7101</td>
</tr>
</tbody>
</table>

Better yet, let’s just plot them in Matlab:

\[
plot(sineTrain(:,1),sineTrain(:,2),'y-',... 
sineValidate(:,1),sineValidate(:,2),'g-',... 
sineTest(:,1),sineTest(:,2), 'c:'); 
legend('train','validate','test'); 
hold on; 
plot(sineTrain(:,1),sineTrain(:,2),'yo',... 
sineValidate(:,1),sineValidate(:,2),'g*',... 
sineTest(:,1),sineTest(:,2),'c+');
\]

I included the results of these commands in Figure 1. I made this postscript figure from within Matlab by doing

\[
print sine\text{-}matlab.ps
\]

after the plot commands.

Now the data has been generated. We are going to try to fit a network to the training data. At some point, we will overfit the training data, resulting in higher error on the validation set. Let’s use one hidden layer with 2, 5, 10, or 20 hidden units and examine the results. Here is the Matlab command and the output it produces:

\[
\text{nnTrain([sinetrain;sinetest],}10\ 10\ 10,1,[2\ 5\ 10\ 20],0.1,1,0.1,0.9,20000,... 
\text{'}c=1 f=sine\text{.results o=long m=scruggs'})
\]

Training with this command:

\[
!( cd /s/parsons/c/fac/anderson/pub/trainvt/nn.dir\text{483101}; train nn.exp>>&..\text{/sine\text{.results &})}
\]

Notice that one of the options is \( f=sine\text{.results} \). This tells \texttt{nnTrain} the file to which the results are to be appended.

Here is the first part of \texttt{sine\text{.results}}, showing the results for 2 hidden units:
Figure 1: Sine data as generated by Matlab
First you see the parameters you specified. Then the errors are printed every 1,000 epochs (this depends on the total number of epochs you specify). Notice that the validation error decreases, then increases. However, the training error continues to decrease. The net begins overfitting about epoch 10,000.

Just how well did this network approximate our sine function? The next output section helps answer this question. It shows a list of the test data patterns by their indices and the desired output (target), the actual output, and the error for that pattern.

Finally, the weights are given at the end of this file so they can be read in by other programs for further analysis of the weights. First the hidden unit weights appear, followed by the output unit weights.

One way to summarize the results is via the `sumshort.awk` awk script, shown below:

```bash
NF == 10 && $1 != "ld.so.1:" {
    ind = "h1 "$1 " h2 "$2 " rh "$3 " ro "$4 " m "$5;
    if (notin(ind))
        indices[numindices++] = ind;
    corr_sum[ind] += $10;
    corr_sqsum[ind] += 10*$10;
    epoch_sum[ind] += $7;
    epoch_sqsum[ind] += $7 * $7;
    rms_sum[ind] += $9;
    rms_sqsum[ind] += $9 * $9;
    num[ind]++;
}
END {
    for (i=0; i<numindices; i++) {
        ind = indices[i];
        printf("%3s %d fc %.3f %.3f RMS %.4f %.4f ep %.1f %.1f\n", ind,num[ind],
            corr_sum[ind]/num[ind],
            confint(corr_sum[ind],corr_sqsum[ind],num[ind]),
            rms_sum[ind]/num[ind],
            confint(rms_sum[ind],rms_sqsum[ind],num[ind]),
            epoch_sum[ind]/num[ind],
            confint(epoch_sum[ind],epoch_sqsum[ind],num[ind]));
    }
}
function notin(ind) {
    for (i=0; i<numindices; i++) {
        if (indices[i] == ind)
```

```bash
```
return 0;
}
return 1;
}

function stdev(sum, sumsq, n) {
if (n > 2)
    return sqrt((n * sumsq - sum * sum) / (n * (n - 1)));
else
    return 0;
}

function confint(sum, sumsq, n) {
    s = stdev(sum, sumsq, n);
    return 1.6449 * s / sqrt(n);
}

function fraction_correct(thresh) {
    n = 0;
    numc = 0;
    while ($1 == "pat") {
        n++;
        if (($4 < thresh && $6 < thresh) ||
            ($4 > thresh && $6 >= thresh))
            numc++;
        getline;
    }
    return numc / n;
}

This script parses the short format of results. I usually combine it with a call to the unix sort command, using a shell script like this:

#!/bin/csh
awk -f /res/nettools/train/summshort.awk $1 |
awk 'printf("%d %d \%.3f %\%.3f %d %\%.3f %\%.3f %\%.1f\n", $2, $4, $6, $8, $11, $13, $16, $19)'
sort -n +6 -7 | more

However, our results are in long output format. So, first let's generate a short version from the long version using the nnlong-to-short.awk script:

$3 == "hi" {
    h1 = $4; h2 = $6; hr = $10; or = $12; mr = $14; maxep = $16;
}

$1 == "epoch" && $2 == maxep {
    trerror = $4;
}

$1 == "Best" {
    ep = $3; valerror = $5; testerror = $7; frcor = $9;
    printf("%d %d \%.3f %\%.3f %\%.3f %\%.3f %\%.3f %\%.4f\n", h1, h2, hr, or, mr, trerror, ep, valerror, testerror, frcor);
}

Again, I usually call this with a shell script like:

#!/bin/csh
awk -f /res/nettools/train/nnlong-to-short.awk $1 |
awk -f /res/nettools/train/summshort.awk |
awk 'printf("%d %d \%.3f %\%.3f %\%.3f %\%.3f %\%.4f\n", $2, $4, $6, $8, $11, $13, $16, $19)'
sort -n +6 -7 | more

If I call this script summlong, then I summarize our results as follows:
The seventh column is the test error, the final column is the best epoch. From this we see that the network with 20 hidden units achieved the lowest test error of 0.068 after the fewest epochs of 2,021.

Now let's see more details of the 20-hidden-unit run. Do this in Matlab with `nnResults`:

```matlab
>> nnResults('sine.results',1)
nnE =
0.2063
nnEp =
10453
Quit, save, or next? (q, s, enter)

nnE =
0.0862
nnEp =
6159
Quit, save, or next? (q, s, enter)

nnE =
0.0708
nnEp =
2349
Quit, save, or next? (q, s, enter)

nnE =
0.0681
nnEp =
2021
Quit, save, or next? (q, s, enter) q
```

I pressed enter after each prompt to get to the final run, the one with 20 hidden units. For each run, two windows are displayed. The window displays for the 20 hidden units run are shown in Figure 2. Figure 2a contains three graphs. The top graph is three learning curves showing RMS error versus epoch, one for each data set. The training error decreases quickly to near-zero error. However, the validation and test data errors decrease below 0.1, but then start to climb. This is the point at which the network is over-fitting the training data. This means that the network is no longer smoothly approximating the curve sampled by the training data and is now beginning to precisely match the training data. This often results in poor interpolation between the training samples resulting in higher error for the validation and test data. We want to keep the weight values near epoch 2,000. This is exactly what is done.

These best weights are used for the next two graphs. The middle graph shows the test data target values and the values predicted by the network using the best weights. There is a pretty close match. Further training would produce less of a match. The bottom graph is a plot of the network's output value versus the target value for the test data. The diagonal represents an exact match between target and output.

Figure 2b is a simple but effective way to visualize the network’s weights. The hidden layer weights can be arranged in a matrix, with columns corresponding to hidden units and rows to inputs. Our network has 20 hidden units, so the hidden layer matrix has 20 columns. We have one variable input and one constant input with a value of 1, so the hidden layer weights form a 2 x 20 matrix. The values are drawn as boxes with positive weights shown as unfilled boxes and negative as filled boxes and their widths represents the weights’ magnitude. The output layer has one unit so its weights form a 21 x
Figure 2: Two Matlab displays for 20 hidden-unit network.
1 matrix, receiving 20 inputs from the hidden units plus a constant input with value 1. Figure 2b displays these two matrices with the hidden layer matrix on top and the output layer matrix below and transposed. Imagine information flowing from the inputs at the upper left of the diagram down through the hidden layer matrix, into the output layer matrix and out to the right.

5 Now what?

You will want to play with the number of hidden units and the learning rates to try to get the lowest testing error possible. Here is how to set up a long experiment for which the number of hidden layers and units and learning rates are varied. This took about 5 hours on a Sun UltraSparcStation.

nnTrain([sineTrain;sineValidate;sineTest],[10 10 10],[2 5 10 20],[1 0 1 0 10],...
[0.01 0.1 1 5],[0 0.9],10000,'c=1 f=sine.results o=short m=scruggs')

Training with this command:
! ( cd /s/parsons/c/fac/anderson/pub/trainvt/mn.dir483101; train nn.exp;& /./././sine.results &)

Now I will use the summshort command to rank the results from best to worst. Here are the first 20 lines of the result:

```
    sumshort sine.results
10 10 10.000 1.000 1 1.000 0.049 4892.0
 0 2 20 10.000 0.100 1 1.000 0.051 10000.0
 5 10 0.100 0.100 1 0.900 0.055 6454.0
 5 10 0.100 1.000 1 1.000 0.057 3400.0
10 10 0.100 1.000 1 1.000 0.059 5777.0
 5 20 10.000 1.000 1 1.000 0.060 5509.0
20 0 1.000 1.000 1 0.900 0.060 2308.0
20 0 10.000 0.010 1 1.000 0.060 3720.0
20 20 0.100 0.100 1 1.000 0.060 4442.0
 5 20 10.000 0.100 1 1.000 0.061 9778.0
20 10 0.100 1.000 1 1.000 0.061 2283.0
 0 2 20 0.100 1.000 1 1.000 0.062 6363.0
 0 1.000 1.000 1 1.000 0.062 6448.0
10 20 1.000 5.000 1 1.000 0.063 5429.0
10 0 1.000 1.000 1 1.000 0.065 9199.0
 5 20 1.000 5.000 1 1.000 0.066 5793.0
10 20 10.000 5.000 1 1.000 0.066 3477.0
20 0 0.100 0.100 1 0.900 0.066 6095.0
10 0 0.100 0.100 1 0.900 0.067 10000.0
 5 0 0.100 0.100 1 1.000 0.068 7368.0
```

The best results were for a network with two hidden layers and 10 units in each layer. Since there isn’t a lot of different in the test RMS error (seventh column) among the top finishers, to draw any conclusions regarding the best network architecture you must repeat this many times. If sine.results contained multiple runs with the same network size and learning parameters differing only in initial weight values, the sumshort will average over the multiple runs.