Introduction to SNNS

Caren Marzban
http://www.nhn.ou.edu/ marzban

Introduction

In this lecture we will learn about a Neural Net (NN) program that I know a little about - the Stuttgart Neural Network Simulator - SNNS, for short. Its homepage is http://www-ra.informatik.uni-tuebingen.de/SNNS/ . Although I have used it for research purposes in the past, it is not the one I'm most familiar with. The ones I use routinely have been written by myself, and are too poorly written to be distributed. I have selected SNNS because it is free, is available for Sun, Linux, and Windows, comes with an extensive manual, and has a reasonably active mailing list. Another one that I have heard some good things about is the Matlab NN toolbox. Both of these have some limitations that I predict will prompt you to look for others, and even write your own. But by then, you'll be knee-deep in NNs and unable to get out. All of which is probably a good thing.

I'm not going to say anything about the installation of SNNS, for I know nothing about that. All I can say is that if you are not good at installing things, then either get someone to do it for you, or heavily utilize the mailing list. After compilation is complete, the two executables we need are xgui and ff_bignet. After you master xgui, I suggest you also try batchman, which is a non-X version of xgui, and therefore much more adaptable.

Jargon

Some of the terminology we have already covered. Number of input nodes is the number of independent variables in the problem. The number of hidden layers and/or nodes is a measure of the nonlinearity of the function, and has to be determined from data; see my previous lectures. Number of output nodes is the number of dependent variables. The activation function is a simple function that sets the value of a given node based on the total input into it. Two common choices are linear and logistic. A linear activation function for all nodes reduces the NN to linear regression. A logistic function is a nonlinear but smoothed step-function bound between 0 and 1. For regression problems (i.e., continuous dependent variables), one usually sets the activation function for the hidden nodes to logistic (so that the NN can learn nonlinear functions), and the activation function for the output nodes to linear (so that the output nodes can approximate any dependent variable). For classification problems (i.e., categorical dependent variables) the output nodes usually have logistic activation

functions. Even for a regression problem, it is possible to have logistic activations for the output node, but then we just have to scale the dependent variable to reside in the range 0 to 1. Either way, it's wise to scale the inputs and outputs to lie in some sane range. See the preprocessing steps in my previous lectures.

The connections between nodes are called weights. Backpropagation is one of the many learning algorithms that iteratively alter the value of the weights until some error function is minimized. Another one - a better one in some sense - is scaled conjugate gradient. Each iteration is called a cycle. All learning algorithms have some number of learning parameters that have to be set by the user. This is another place where some trial and error is necessary. The error function being minimized also depends on the user; see my performance assessment lecture. For regression problems, if you don't know what error function to select, then go for the mean squared error. Every one else does. For classification problems, it's natural to set the error function to something called cross-entropy, because then the output of the NN (if all the activation functions are logistic) is the (posterior) probability of belonging to a category, given the values of the inputs. This probability is exactly the kind of probability one needs for forecasting purposes, because it is conditional on the input data. By the way, here is one defect of SNNS - to the best of my knowledge, it does not have a built-in option for setting the error function to cross-entropy. Maybe by now it does. But if it doesn't then you'll just have to use mean squared error, or dabble around with the snns kernel itself. Either way, you may want to be more cautious about betting on the meaning of the outputs as probabilities.

In minimizing the error function, the learning algorithm can get caught in a local minimum. A local minimum is a set of values of the weights that yield a minimum of the error function, but not necessarily the lowest value. Actually, it's not necessary to find a global minimum, but it is important to find a sufficiently deep local minimum. There are lots of ways to handle this problem of local minima, but at least you should try training from different initial weights to see what sorts of local minima you get.

The data is usually divided into a training set (for estimating the weights) and a validation set (for estimating the optimal number of hidden nodes). Actually, we need several validation sets to assure we are not overfitting the validation set itself. And finally, we need a test set for getting an unbiased estimate of performance. But, here we will use one training set and one validation set.

Preparation

We need three files before we can run things; one "network file", and at least two "pattern files". Let's call them 1) reg.net, 2) trn.pat, 3) vld.pat . As suggested by the names, we are trying to develop a network for performing nonlinear regression (reg.net), based on a training set (trn.pat) and a validation set (vld.pat).

The format of these files is crucial. The easy ones are the pattern files. They all start with the following:

SNNS pattern definition file V3.2
generated at Mon Apr 25 18:08:50 1994
(followed by 2 blank lines)

No. of patterns : 9
No. of input units : 2
No. of output units : 1
(followed by 1 blank line)
(followed by the data)

For 2 inputs and 1 output, the data should be in the form

x1 x2
target
x1 x2
target
etc.

The tough one to make is the network file. SNNS gives you many ways of making it, but I like one that employs ff_bignet, which makes a file called SNNS_FF_NET.net . This is the file we will rename reg.net.

Here are a few examples that will tell you how to use it:

ff_bignet -p 1 2 -p 1 4 -p 1 1 Act_IdentityPlusBias -l 1 + 2 + -l 2 + 3 +

will make an NN with the following attributes:
Number of input nodes = 2
Number of hidden nodes on layer 1 = 4
Number of output nodes = 1
Activation function for hidden layer nodes = logistic (default)
Activation function for output layer nodes = linear
Each layer fully connected to adjacent layers.

ff_bignet -p 1 17 -p 1 15 Act_IdentityPlusBias -p 1 3 Act_IdentityPlusBias -p 1 14 Act_IdentityPlusBias -p 1 5 Act_IdentityPlusBias -l 1 + 2 + -l 2 + 3 + -l 3 + 4 + -l 4 + 5 +

will make an NN with the following attributes:
Number of input nodes = 17
Number of hidden nodes on layer 1 = 15
Number of hidden nodes on layer 2 = 3
Number of hidden nodes on layer 3 = 14
Number of output nodes = 5
Activation function for all nodes = linear
Each layer fully connected to adjacent layers.

ff_bignet -p 1 17 -p 1 15 -p 1 3 Act_IdentityPlusBias -p 1 14 -p 1 5 Act_IdentityPlusBias -l 1 + 2 + -l 2 + 3 + -l 3 + 4 + -l 4 + 5 +

will make the same network but with logistic activations for the first and third hidden layers. It's unlikely that you'll need more than one hidden layer of nodes, but this way at least you'll know how to set up the NN if you do need more hidden layers.

You've probably deduced that the -p statements sets the number of nodes on each layer, and the -l statement does the task of connecting the various nodes. For the complete description of these commands, see the SNNS manual (page 270), or the man pages for ff_bignet, which can also be found on the web.

Example

Let's *make* some data. This is the C code I used to make the data:

```
main()
{
int i,j;
double x,y,target;
FILE fp;
fp=fopen("data","w");
for(i=0;i ≤ 20;i++){
x = (double) i/10 -1;
for(j=0;j ≤ 20;j++){
y = (double) j/10 -1;
target=xx+yy;
fprintf(fp,"%lf %lf %lf \n",x,y,target);
}
}
fclose(fp);
return 0;
}
```

In other words, the data consists of 2 independent variables, x and y, varying from -1 to +1, and a dependent variable (target) that is simply a paraboloid over the x-y plane. This is an example of noiseless data, for the target is a deterministic function of x and y, but it will serve to illustrate SNNS. I encourage you to add a little noise to the data (like target=x*x+y*y + 0.1*drand48();) and see how that affects the performance of the NN.

We are not going to feed all of this data into the NN. Instead we will train an NN based on only 9 of the cases, and validate on a different 9 cases. In particular, trn.pat looks like this:

SNNS pattern definition file V3.2
generated at Mon Apr 25 18:08:50 1994

No. of patterns : 9
No. of input units : 2
No. of output units : 1

-0.800000 -0.800000
1.280000
-0.800000 0.000000
0.640000
-0.800000 0.800000
1.280000
0.000000 -0.800000
0.640000
0.000000 0.000000
0.000000
0.000000 0.800000
0.640000
0.800000 -0.800000
1.280000
0.800000 0.000000
0.640000
0.800000 0.800000
1.280000

The file vld.pat looks like this:

SNNS pattern definition file V3.2
generated at Mon Apr 25 18:08:50 1994

No. of patterns : 9
No. of input units : 2
No. of output units : 1

-0.400000 -0.400000
0.320000
-0.400000 0.000000
0.160000
-0.400000 0.400000
0.320000
0.000000 -0.400000
0.160000

0.000000 0.000000
0.000000
0.000000 0.400000
0.160000
0.400000 -0.400000
0.320000
0.400000 0.000000
0.160000
0.400000 0.400000
0.320000

Now, let's make a network with 2 input nodes, 4 hidden nodes (on one layer), and one output node with a linear activation function:

ff_bignet -p 1 2 -p 1 4 -p 1 1 Act_IdentityPlusBias -l 1 + 2 + -l 2 + 3 +
As I said, rename SNNS_FF_NET.net to reg.net .

Running SNNS

Now that we have made our 3 files - reg.net, trn.pat, vld.pat - we can train and validate a NN.

Run xgui
Left click on logo to make it disappear
In the Manager Panel click on FILE
Double click on item called reg (This puts reg into the type-in window), and press LOAD
Click on Patterns
Double click on item called trn, followed by LOAD
Double click on item called vld, followed by LOAD
Click DONE

We have now loaded the network file and the two pattern files.

In the Manager Panel click on CONTROL
For CYCLES use 200
For VALID use 1 (any number other than 0 will allow multiple pattern files, like training and validation files)
From the top SEL. FUNC list select SCG (Scaled Conjugate Gradient)
For LEARN use 0.001 0.01 blank blank
Click on the top USE and select trn
Click on the bottom USE and select vld

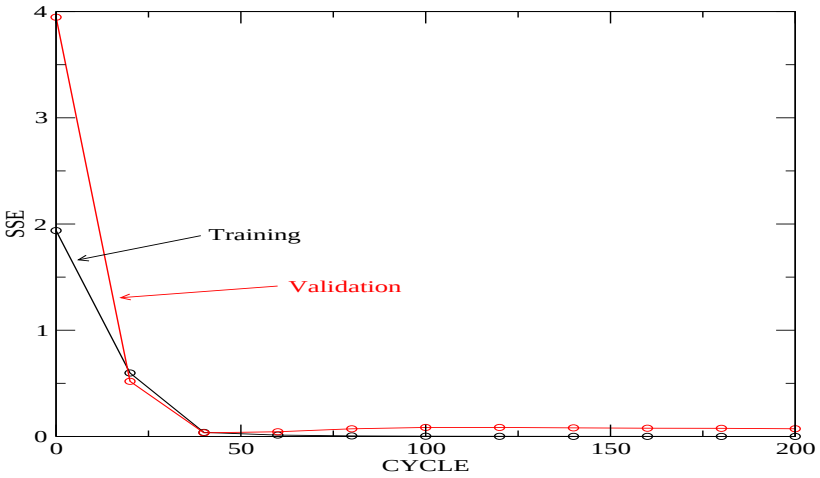
Back in the Manager Panel, click on GRAPH in preparation for viewing the training

and validation error at each training cycle.

Now, sit back and look at what we just did in the control panel to see what we are trying to do. It should be relatively clear. If it's not, it will be.

Finally,
Click on INIT (to initialize the network weights)
Click on ALL

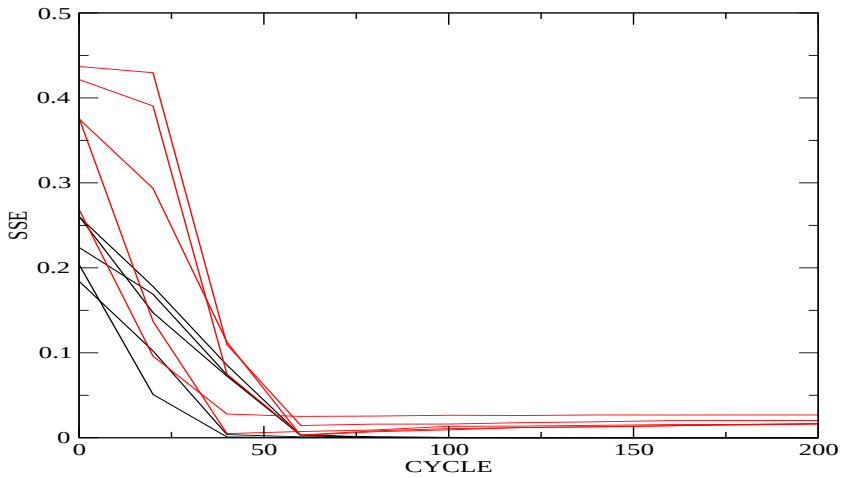
This will start the training. Some error values will be printed on the screen, and a more detailed plot of the training and validation error will appear in the graph window. It will look something like this, but only a lot finer:



You can see that as training progresses (through the cycles) the training error reduces monotonically. This is expected and implies that the NN is learning the underlying function. In fact, eventually it learns it perfectly, i.e., with zero error. However, you must recall the phenomenon of overfitting (see my previous lectures). The purpose of the validation set is to guard against overfitting. You can see that the validation error falls off too, at least initially. But after about 50 cycles, it begins to grow. This is the onset of overfitting. In this case overfitting is occurring because we allowed the network to have 4 hidden nodes. This amounts to $2 \times 4 + 4 \times 1$ weights and $2 + 1$ biases, for a total of 15 parameters that must be determined from only 9 data cases. Think of 9 equations and 15 unknowns. If we had 9 unknowns and 9 equations, then we could solve the set exactly, i.e., with zero error. With 15 unknowns we can do even better, if it were possible to have less than 0 error. The point is that the network

has too many parameters and is therefore too flexible. I suggest you try different number of hidden nodes and repeating the above steps to see what happens. Follow the procedure in my previous lectures for arriving at an optimal number of hidden nodes.

Let's also give a little thought to the matter of local minima. To that end,
Click on INIT
Click on ALL
Watch the error graph. Do this INIT/ALL repeatedly to see how the error curves change depending on the initial values of the weights. You'll get something like this:



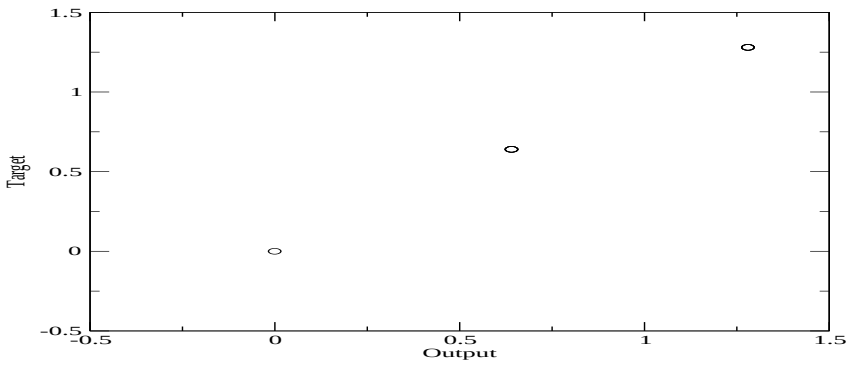
Note that there is quite a bit of shifting around of the training and validation curves as we do the INIT/ALL sequence repeatedly. In spite of the huge variations, certain patterns occur more frequently than others. In a sense, it is the average of all of the curves that should be examined for determining, say where training should be halted (i.e., before overfitting sets in).

In Manager Panel, if you click on FILE, you'll get the file browser window again. Click on Network. Now you can put a new name in the window and click SAVE to save the weights (and more) of the trained network. I do this when I need to read in the weights into one of my own codes for doing, say, performance assessment.

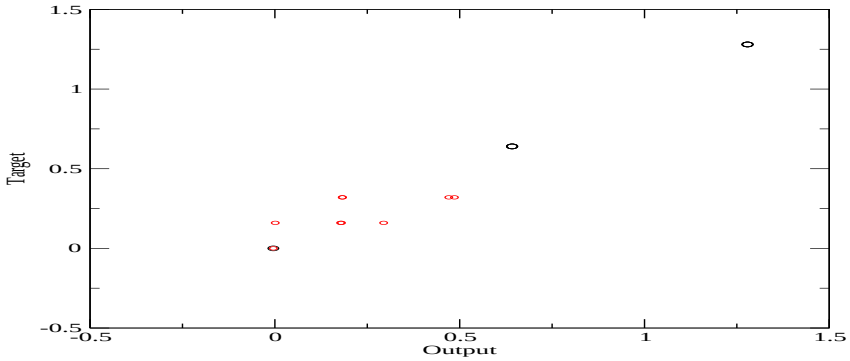
Suppose you want to see the output values of the output nodes. In the file browser window, click on RES and put a new name in the window, and then click SAVE. This will bring up another window with a few self-explanatory buttons in it. Say YES to everything, followed by DONE. Although the exact output values that you will get

on your computer may be different from the ones I get, the resulting file will look something like this:

Bla bla bla
#1.1
-0.8 -0.8
1.28
1.27887
#2.1
-0.8 0
0.64
0.64101
#3.1
-0.8 0.8
1.28
1.27954
...
You can now see that when the inputs are -0.8 and -0.8, the output of the trained NN is 1.27887. This can be compared to the actual target value of 1.28. Similarly, when the inputs are -0.8 and 0, the output is 0.64101, as compared to the target value of 0.64. Not bad! In fact, if you plot the target (i.e., actual) values versus the output (i.e. predicted) value, you'll get the following scatterplot:



A diagonal line is good. It means we have perfect prediction at least when we test the NN on the training set itself. But how about the validation set. Well, I don't know how to get snns to print a similar result file for the validation set. For that I read the weight values from the result file into one of my own codes, which in turn allows me to do anything I desire. Well, almost everything. At least, I can get the scatterplot for the validation set. In fact, here it is for both the training and validation sets



You can see that the NN learns the training set perfectly, but it learns the validation set with some error. At least the points are scattered close to the diagonal. See my performance assessment lectures to see how to assess the performance of this NN.

After you have played around enough, close any window from the X in the top right corner, and the whole thing will shut down. Actually, I think of this as a nuisance, but snns presents it as a feature.