CS 594 — Neural Networks

Back-Propogation

April 21, 1998

Turn in your hard-copy writeup for this lab by April 30 if possible, but by May 11 at latest.

(A copy of this lab writeup is contained in
`http://www.cs.utk.edu/~mclennan/Classes/594-S98/proj-2.ps`)

1 Preliminaries

In this lab you will train a back-propogation algorithm to perform a non-trivial function under various circumstances and evaluate the results.

For this lab you will be using the bp program from the pdp software package. You will also need files available in another directory. To get the files, do the following:

```
mkdir ~/cs594/bp
cp /mdx/usr/local/pdp/bp/* ~/cs594/bp
cp ~mclennan/pub/594/proj2/* ~/cs594/bp
```

NOTE: Executables will not run on Sun3's. All graphs that are to be turned in with this lab should be labeled appropriately.

2 Introduction

There is a summary of basic pdp commands at the end of this lab. The network that will be used will have two inputs, two hidden layers with five units in each, and one unit in the output layer. The function that we are training on is the sum of two gaussian curves in three dimensions and is displayed on the next page. The input is plotted on the x and y axes and output on the z (vertical) axis.

3 Tools

3.1 bp

When you first start **bp**, (e.g. bp gauss.tem gauss.str) you will see several fields on the screen. Note: this program requires a screen with more than 24 rows to display correctly. If you do not have this capability, you may use an alternate template, `gauss2.tem` which does not print the weights to the screen and will therefore fit on a 24 line screen.

The *tss* value is the total sum of squares of the error, i.e. $\sum_p (t_p - o_p)^2$. The *epoch* value lists the number of epochs run. This value is initially set to zero and is reset with the newstart commmand. If you load a set of partially trained weights, remember that the number of epochs displayed will only reflect training after loading. You can change the number of epochs displayed by entering `set state epochno` so that it reflects the true number of epochs run on the weights.

All other entries on the screen are represented in hundredths, so 43 means 0.43 and 124 means 1.24. Negative numbers are presented in reverse video. There are two inputs, two hidden layers with five units each, and one output unit in the last layer. The target value is presented next to the output value for comparison. The three weight matricies are then given with columns representing the 'from' units and the rows representing the 'to' units.

You will be able to easily alter the learning rate and other parameters at the command prompt in the program. The two things you will be most interested in changing are the learning rate and the number of epochs to train on at a time. To change the learning rate, enter `set parameter lrate`. You will be reminded what the current value is and if you decide not to change this value you can simply hit return. To change the number of epochs to train at a time, enter `set nepochs`. You will be reminded of the old value and then prompted to change it. To change the value for which training will end, enter `set ecrit` and then enter the new value. Commands can also be shortened to the point where ambiguity between commands does not exist, i.e. `set p l` will do the same as `set parameter lrate`. Any menu choice with subchoices is followed by a forward slash, so if for example you type `set` and then return, you will be prompted with a list of items to continue with. If you do not wish to continue, you can type return to get back to the main menu. The values loaded initially are lrate = 0.08, nepochs = 50, and ecrit = 0.04.

To save weights enter `save w` and you will be prompted for a filename. Since you will likely be creating many files, you may want to have your filenames include the number of epochs and the tss value at the time of saving. For example, you might name a file `w_3_2400_2.3241` to mean a weight file of your third experiment after 2400 epochs at a tss of 2.3241.

You will be supplied with several pattern files. Each line in a pattern file has 4 entries: the name of the pattern, the x value, the y value, and the output or z value. For this lab the name of each pattern is the number of the pattern in the file. The naming convention for the pattern files is `g`*number-of-patterns*`.pat`.

To create a data file from within the bp program, enter the command `do gauss.do`. This will cycle through all the patterns testing each one and printing the results in a file called `out1` (which you will use with the program error_file to create the datafile for gnuplot). Note that calling `gauss.do` will append to an already existing copy of `out1`, so you will need to rename or delete `out1` before you enter `do gauss.do`.

One of the main purposes of this lab is to notice how training is proceeding by viewing how the tested output of the weights changes over training.

3.2 Gnuplot

Before using **Gnuplot**, you will need to add the following to your path in your `.cshrc` file:

```
/usr/local/X11R5/bin
```

This will make the program available from your directories. **Gnuplot** is available on the hydra and cetus machines. Although **Gnuplot** has shortcomings, it will be able to do the 3d graphs needed for this lab.

Gnuplot has online help and is capable of plotting functions using the same commands that are found in the C programming language. To do a 3d plot on a data file, you must set the program into parametric mode with the command `set parametric`, and then you can plot the file with the command `splot 'data' with lines` where `data` is the name of your data file. Your data file should contain three columns where each line contains three points which represent the x, y, and z coordinates. Instead of plotting with lines, you may also plot with points, dots, or linespoints. Once a style is set you only have to enter `splot 'data'` and the last defined type will be used. **Gnuplot** also allows you to title graphs with commands like: `splot 'data' title '2500 epochs'`.

To plot a graph on the screen:

```
set parametric
splot 'error1' with lines
```

To print a graph to the cslab printer:

```
set parametric
set term postscript
set output '|lpr -Pcslab'
splot 'error1' with lines
```

To change back to the screen after printing, enter `set term x11` and `replot`.

3.3 Error Graphs

After you have created a data file and graphed it, you may want to know how your data differs from the target values. A program has been written for you which will take as input your data file and create a new data file which has as it's z-axis the square of the difference of the target value and the output value. On a perfectly trained set, this error file would graph as a flat surface of height zero. Where there are humps in the graph is where your program is producing an incorrect answer and the height represents how much error your network is producing. The name of the program is **error_file** and takes as an argument the data file you created. It will output to a file called `error1` which is ready to be graphed by **Gnuplot**.

4 Experiments

The training set has 121 patterns which evenly cover the domain. Your goal will be to minimize the total sum of squares error. Start the bp program by entering `bp gauss.tem gauss.str`. This will load a template file which describes the layout of the output, and gauss.str contains commands that are given before training is performed.

You may wish to modify this file to initial values you find more appropriate, or simply change the values after the program has begun.

To train the network, type ptrain (or p) or strain (or st). ptrain stands for permuted training and mixes the inputs so that there is not an ordering bias. strain stands for sequential train and processes the patterns in the order encountered in the training file. Because we will be using batch learning there will be no effective difference between the two. The screen is updated only after all patterns have been processed. The program will run through 50 epochs unless you change the value of *nepochs* and you will see the weights change as patterns are processed. Hitting `Ctrl-C` will stop a training run and you can then choose whether to continue or break.

1. Run the ptrain command and observe the results. When the 50 epochs are complete, enter `newstart` and repeat the experiment at least 4 times. On most of these, you should notice something peculiar about the output unit and the weights. What is it and why is this happening?

2. Train the network until it has achieved the performance defined by $ecrit = 0.04$ (training will stop automatically if this is attained). This should be possible without too much trouble and should happen fairly soon (around 7000 epochs). If the network is having trouble you can enter the *newstart* command to start over again with a different set of random weights. Notice roughly how long it takes and how the error changes over time. Now repeat the training and save the weights at five roughly equally spaced times in a successful training (the last time you save weights should be after learning has stopped). An easy way to do this is to save weights often and then when training stops you can use the weight files that best represent the training process. Make a note for yourself of how many iterations were performed and what the *tss* value was at each time.

   To get an idea of how well the net performs on these five sets of weights, we will graph the functions that these weights are actually computing. To do this, we will test the performance of the weights on a very large set of patterns and graph the results. To create the data files for each set of weights,

   ```
   get pat g2500.pat
   get weight your_weight_file
   do gauss.do
   ```

Each invocation of **gauss.do** will create the file `out1` if it doesn't exist or will append to it if it does exist. View each data file using **Gnuplot** and print the results as described above. This gives you a "movie" view of how your function is evolving. Attach your printed and labeled graphs. When you are done with this part of the lab, either exit and restart the **bp** program or load the `g121.pat` file and type `newstart`.

3. Your final graph should look very similar to the true function it is trying to simulate. To find out how good it is, we will graph the sum-squared error. The program **error_file** described above will produce a data file you can graph to see the error. **Attach this graph and make observations in the space below.**

4. How does this graph relate mathamatically to tss?

5. The network you just trained has bias units. To remove them, edit `gauss.net` by changing the line `%r 2 11` to `%r 2 0`. Train the net for at least twice as long as you did for the successful run above (but don't expect success). Discuss your results below. When done with this problem, change `gauss.net` back to it's original state. Also what was the final tss value?

6. In order to find out what kind of feature detectors the second hidden layer is developing, we will plot the output of each of these units for all possible inputs. Quit the **bp** program and edit the `feature.str` file so that "`your_fully_trained_weights`" is replaced by the name of the file that has the fully trained weights from a previous excercise. Then at the unix prompt, type `bp feature1.tem feature.str`. Create a data file by entering `do gauss.do` and graph this and the other features (using template files `feature[1-5].tem`) and attach the graphs. Comment below on what these features are and how they are working together to produce the desired function.

7. To get an idea of how training is progressing, we will plot a 2d graph of epochs vs. tss. Start the **bp** program with `bp epoch_vs_tss.tem epoch_vs_tss.str`. Type `do evt.do` to start the process. It will train for 3,000 epochs or until the error criteria is met. The learning rate is lower on this experiment (0.02) to help ensure convergence. The output goes to `evt1` and can be graphed with the `plot` command (as opposed to `splot`). Attach at least one such graph and comment on it below.

8. Epochs cost money. Restart the program and try to minimize the number of epochs by adjusting the learning rate at various times during the training. Report your results and describe your strategy. You will need to experiment with this some; report failures as well as successes.

Other Useful Commands

quit — Quits from the program (asks for confirmation).

reset — Resets activations to resting level and the weights to their previous starting values (i.e., does not randomize them).

strain — Sequential training; input patterns are presented in the order in which they occur in the `.pat` file.

test — Tests the network on a single input pattern from the pattern list. Prompts for pattern name or number.

log — Open or close a log file.

run — Pass a command to the shell for execution. The command is terminated with `end` or an extra return.

set — Examine and set control variables, activations, weights, modes and other parameters.

disp — Allow user to display information on the screen or change display options.

get — Load files containing patterns, weights, etc.

save — Save files containing weights or screen contents.

Note: If you plan on using the PDP software very much, you should add the following to the path in your `.cshrc` file:

`/usr/local/pdp/bin`

Then you will not have to change to the `/usr/local/pdp` directory to run the programs. However, any files that the simulation needs, such as the `.str` or `.loo` files, will have to be copied to your local directory (which is what you would have to do anyway if you were planning on changing them).

The *Explorations* book is very useful if you want to learn about neural networks. It costs about $35 and includes the software. You can buy either a Mac or an MS-DOS version; both include source files.