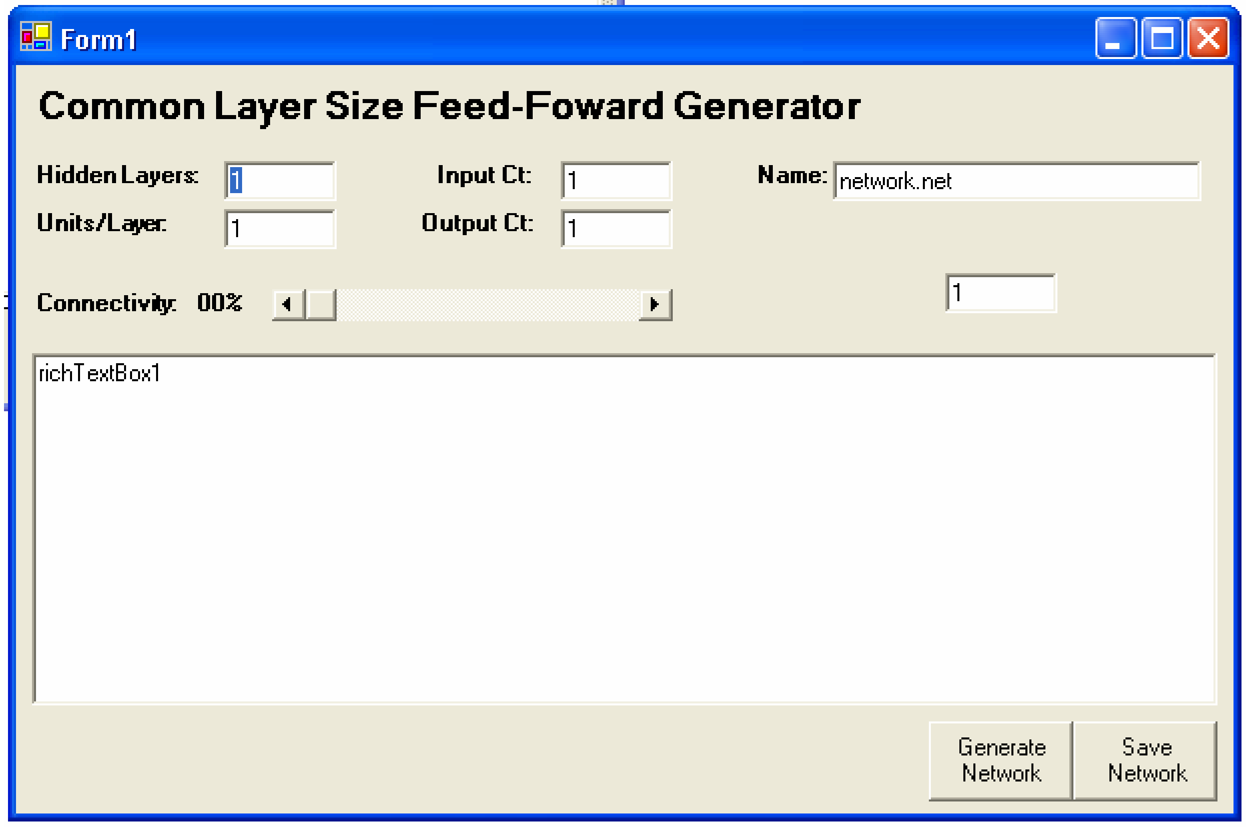
10-17-04

Work has concentrated on developing the program needed to generate the networks for SNNS. Because the speed of SNNS' graphical network display is extremely slow as the network gets large, I have decided it is not practical to attempt designing the networks within SNNS. Thus the C# program is being developed. To begin, I am writing the code to simply create a feed forward network of any number of layers, with a specific width and number of inputs ands outputs…following the structure of the SNNS network file format. The interface is shown as figure 1.

Figure 1: Network Generator Interface



Form1

Common Layer Size Feed-Foward Generator

Hidden Layers: 1

Input Ct: 1

Name: network.net

Units/Layer: 1

Output Ct: 1

Connectivity: 00%

1

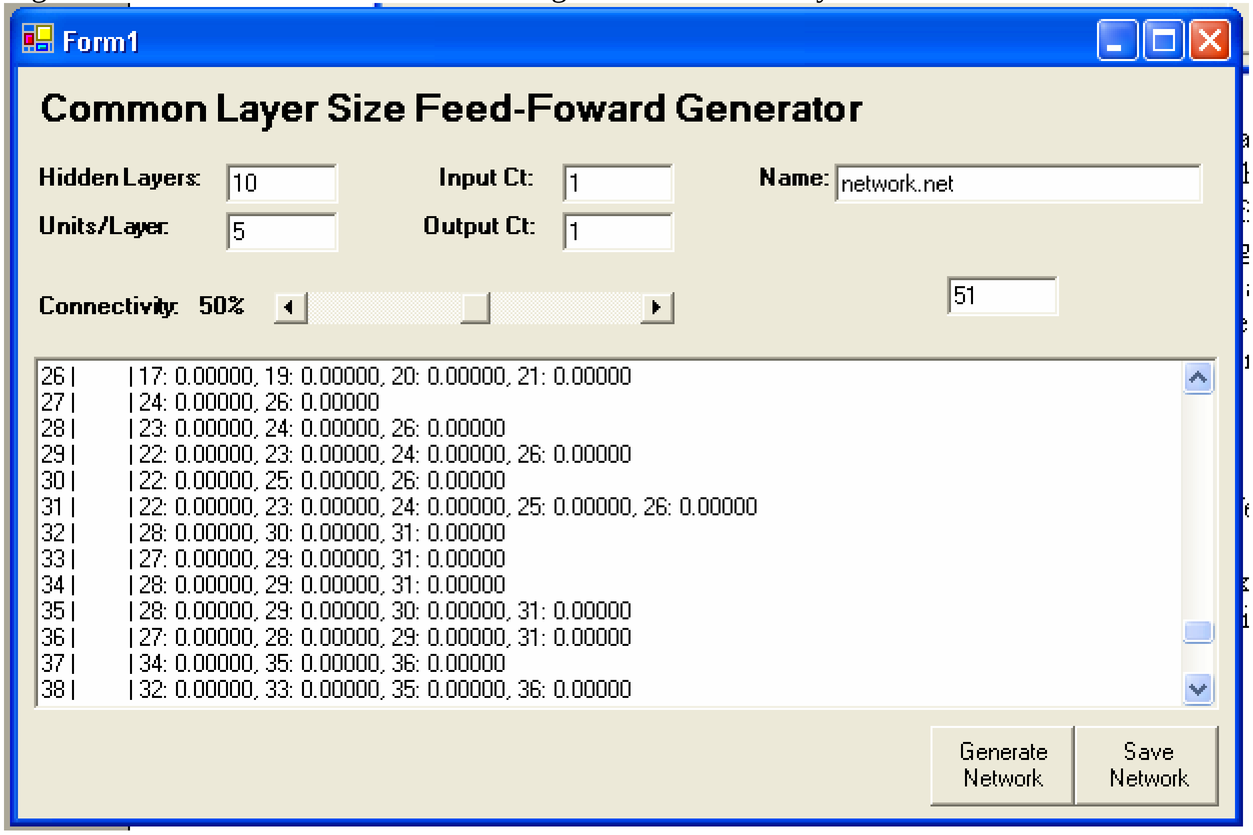
richTextBox1

Debugging has gone fairly smoothly, the main issue being that having a network with layers more than about 50 nodes tends to become extremely slow. The problem was using the append member of the richTextBox class to add to the output file. I have found it is better to create a string for each new line and then insert the string at one time into the textbox. The code for the program up to this point is viewable as "NetGen1" in the Journal Files folder. At this point, the next step is to implement the connectivity feature. Currently, the program produces an output file which is a fully connected feed forward network.

10-21-04
I will work today on finishing up the network generator for feed forward networks and also perfect the process to convert PGN games file into EPD, and then into arrays which may be used in SNNS training and verification files. I will explain the files when I get to this point, but for now want to finish the program shown first in figure 1.

After some difficulty in getting the Random class to function correctly, I am able to produce an acceptable output file. A screen shot of the new application is shown in figure 2. The output file format is based on "test.net" located in the Journal Files folder. I expect to see nodes in the output file with varying source nodes recorded…this is actually observed very well in figure 2. A fully connected network would have matching nodes for any given layer.

Figure 2: New Network Generator Showing Partial Connectivity Network



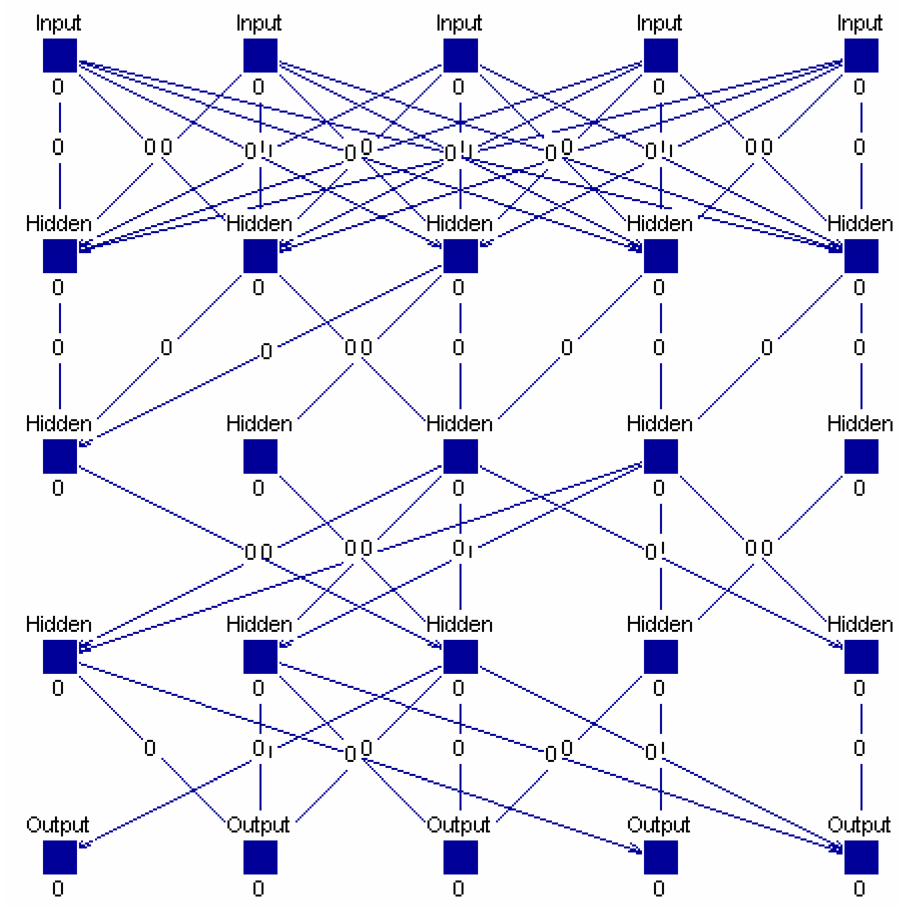
Hidden Layers: 10
Units/Layer: 5
Input Ct: 1
Output Ct: 1
Name: network.net
Connectivity: 50%
51

```
26 |      | 17: 0.00000, 19: 0.00000, 20: 0.00000, 21: 0.00000
27 |      | 24: 0.00000, 26: 0.00000
28 |      | 23: 0.00000, 24: 0.00000, 26: 0.00000
29 |      | 22: 0.00000, 23: 0.00000, 24: 0.00000, 26: 0.00000
30 |      | 22: 0.00000, 25: 0.00000, 26: 0.00000
31 |      | 22: 0.00000, 23: 0.00000, 24: 0.00000, 25: 0.00000, 26: 0.00000
32 |      | 28: 0.00000, 30: 0.00000, 31: 0.00000
33 |      | 27: 0.00000, 29: 0.00000, 31: 0.00000
34 |      | 28: 0.00000, 29: 0.00000, 31: 0.00000
35 |      | 28: 0.00000, 29: 0.00000, 30: 0.00000, 31: 0.00000
36 |      | 27: 0.00000, 28: 0.00000, 29: 0.00000, 31: 0.00000
37 |      | 34: 0.00000, 35: 0.00000, 36: 0.00000
38 |      | 32: 0.00000, 33: 0.00000, 35: 0.00000, 36: 0.00000
```

It is clear in the textbox in figure 2 that the nodes are only partially connected at this time. The key to getting Random to work is to declare a new object of type random at the top of the network generation routine, and not inside a loop. Each time the new class is created, the seed is apparently the same, so we end up getting duplicate nodes, which is obviously not desired. The new version of the code may be seen as "NetGen2" in the Journal Files folder.

Figure 3 shows the types of networks this application is designed to create. It is a screen capture from SNNS. It must be noted that the first hidden layer is ALWAYS fully connected to the input layer, but all following layers are partially connected in some random configuration.

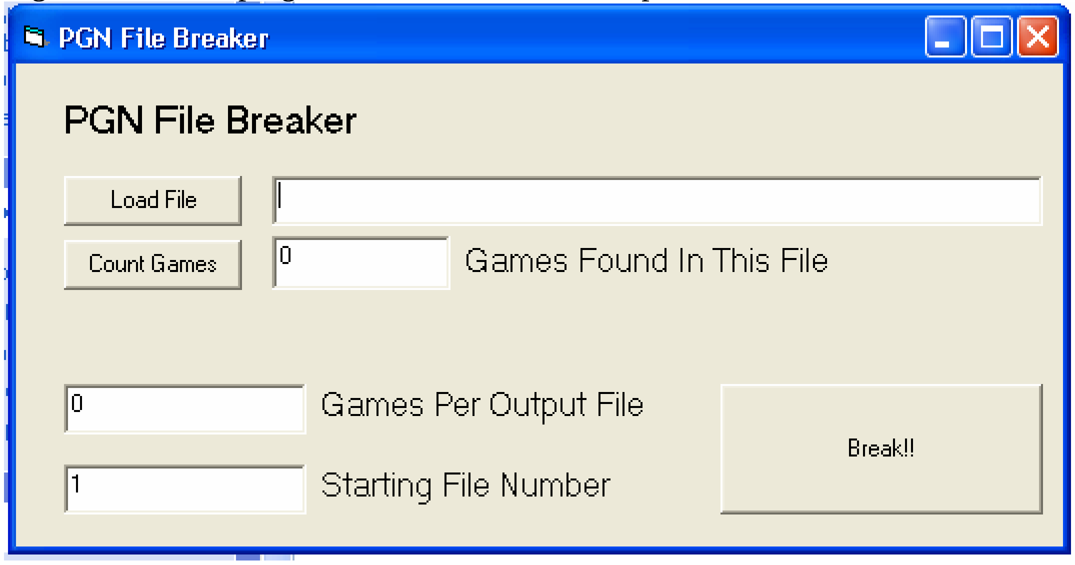
Figure 3: Network Architecture created by NetGen



With the network generator now fully functional, it is time to start considering the data file processing. The dataset I will be using is capable of producing PGN (portable game notation) files, which are algebraic, standardized Chess game recordings. ChessBase 9.0 (NEED TO ORDER!!) should be able to provide a few million games for use in this project, so the amount of data is obviously massive. An efficient data processing method is therefore required. A document describing the PGN standard is provided in the Sources folder (Pgn.pdf).

I decide to use a program I find on www.pgn.freeservers.com in order to convert the PGN files to EPD files. This application is called PGNposition and is a command line utility. The PGN file must be specified, and an output EPD file must be supplied at run time. Unfortunately, the utility is very sensitive to errors in the PGN files…If it comes across one, it seems to crash. Rather than writing a new conversion utility (not very easy), I decide to instead write a program which will break the larger PGN database files down into smaller files to be processed one at a time. This way, an error in one PGN game will not cause a great deal of failed conversions, and can possibly be found and easily corrected. This program is called "Breaker" and a screenshot may be seen in figure 4.

Figure 4: Breaker program screen shot…used to split PGN files



PGN File Breaker

Load File	
Count Games	0 Games Found In This File
Games Per Output File	0
Starting File Number	1

The PGN Breaker program is written in Visual Basic 6.0, and is simply a text parser. The code is shown in the Journal Files Folder as "BreakerCode." An example of the PGN format is shown in figure 5.

Figure 5: PGN Format example

```
[Event "Hastings 8081"]
[Site "?"]
[Date "1980.??.??"]
[Round "01"]
[White "Liberzon,Vladimir"]
[Black "Chandler,Murray"]
[Result "1-0"]

1. e4 d6 2. d4 Nf6 3. Nc3 g6 4. Nf3 Bg7 5. Be2 O-O 6. O-O Bg4 7. Be3 Nc6
8. Qd2 e5 9. d5 Ne7 10. Rad1 Bd7 11. Ne1 Ng4 12. Bxg4 Bxg4 13. f3 Bd7 14. f4
Bg4 15. Rb1 c6 16. fxe5 dxe5 17. Bc5 cxd5 18. Qg5 dxe4 19. Bxe7 Qd4+ 20. Kh1
f5 21. Bxf8 Rxf8 22. h3 Bf6 23. Qh6 Bh5 24. Rxf5 gxf5 25. Qxh5 Qf2 26. Rd1
e3 27. Nd5 Bd8 28. Nd3 Qg3 29. Qf3 Qxf3 30. gxf3 e4 31. Rg1+ Kh8 32. fxe4
fxe4 33. N3f4 Bh4 34. Rg4 Bf2 35. Kg2 Rf5 36. Ne7 1-0
```

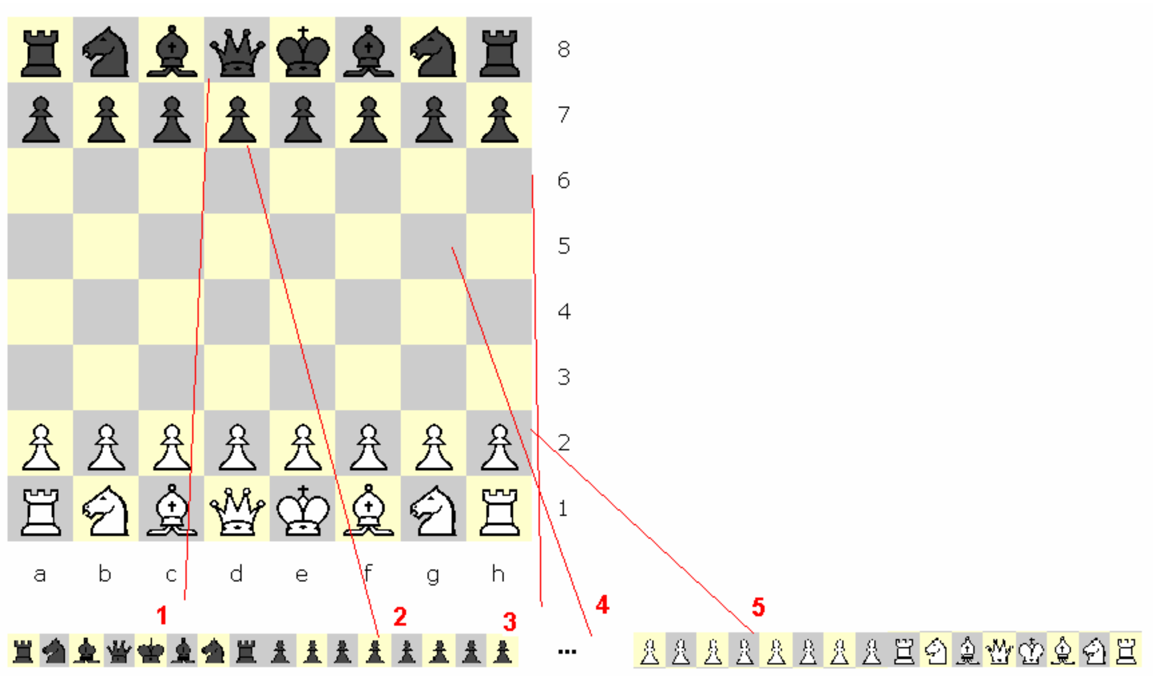
EPD notation is "expanded position description" and is also a standard, although not nearly as popular as the PGN notation. PGN is far more compressed as it does not record a complete board description for each move as EPD does. EPD consists of a string for each move in the game, a typical example of which is shown in figure 6.

Figure 6: EPD File Example

```
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - pm d4;
rnbqkbnr/pppppppp/8/8/3P4/8/PPP1PPPP/RNBQKBNR b KQkq d3 pm Nf6;
rnbqkb1r/pppppppp/5n2/8/3P4/8/PPP1PPPP/RNBQKBNR w KQkq - pm Nf3;
rnbqkb1r/pppppppp/5n2/8/3P4/5N2/PPP1PPPP/RNBQKB1R b KQkq - pm b6;
rnbqkb1r/p1pppppp/1p3n2/8/3P4/5N2/PPP1PPPP/RNBQKB1R w KQkq - pm g3;
rnbqkb1r/p1pppppp/1p3n2/8/3P4/5NP1/PPP1PP1P/RNBQKB1R b KQkq - pm Bb7;
rn1qkb1r/pbpppppp/1p3n2/8/3P4/5NP1/PPP1PP1P/RNBQKB1R w KQkq - pm c4;
rn1qkb1r/pbpppppp/1p3n2/8/2PP4/5NP1/PP2PP1P/RNBQKB1R b KQkq c3 pm Bxf3;
rn1qkb1r/p1pppppp/1p3n2/8/2PP4/5bP1/PP2PP1P/RNBQKB1R w KQkq - pm exf3;
rn1qkb1r/p1pppppp/1p3n2/8/2PP4/5PP1/PP3P1P/RNBQKB1R b KQkq - pm e6;
```

Where p is pawn, K is king, etc. Black is lowercase and white is uppercase. It is obviously required to take the EPD files and convert them one more time, this time into input vectors to be used by the training mode (in SNNS). Figure 7 demonstrates how the EPD file is generated, by taking each row of the chess board and merely placing them next to each other.

Figure 7: EPD Format and how it is generated from the board



The inputs into the neural network will be in the same order as the positions are arranged for EPD format. Because the inputs to the network must be floating point values between

+1 and -1, I decide to assign values based on the traditional weights given to the pieces in the game. Black will acquire + values, and white will acquire -. Figure 8 shows the weights which will be assigned, based on the character present in the EPD file. A program will be created shortly which will convert the EPD strings into floating point vectors (training data sets).

Figure 8: Weights assigned for each piece

Piece	EPD Char	Weight
King	k,K	1.0,-1.0
Queen	q,Q	0.9,-0.9
Rook	r,R	0.5,-0.5
Knight	n,N	0.4,-0.4
Bishop	b.B	0.3,-0.3
Pawn	p,P	0.1,-0.1

Typically, the knight and the bishop are each given a weight of 3, but there is a need to differentiate these pieces in the input vector, so I decide to assign the knight .4, slightly more "valuable" than the bishop. However, these "values" may not actually have any meaning to the NN once it is training, and seem more likely to serve as "placeholders" than anything else.

The program will be created in C#, once again it is little more than a string parser. The EPD file will be opened, and each character in the description string must be converted to a numeric character according to figure 8. Two more important requirements must be met:
-The program must also produce the "next move" for the player to make, and save only BLACK TO MOVE positions.
-The output file must be compatible with SNNS (the data file format rules must be followed).

The format requirements for the SNNS files may be seen in the file "SNNSPattern.pat" located in the Journal Files Folder. Essentially, a header must specify how many inputs and outputs we have, as well as the total number of patterns to be found in the file~~. It is important to realize that eventually, the move must be replaced by some integer value~~ for the geographical representation of the game (which will be examined first). The strings will eventually be classified based on the next move to be made (highlighted in figure 6) so that the output may be specified as a zero or a one for training (0 means don't make the move, while 1 will 'make it'). See the functional description for more details regarding the geographical representation of the game.

For now, I will just keep the move to be made in algebraic chess notation. ?? Is this the best way to do this?

10-28-04

ChessBase 9.0 has been ordered. I am waiting for it to arrive so I can complete work on the data processing programs. For now I will be working on generating networks of various dimensions and trying to train them with sample data. This is being done in order to come up with an estimate of how long it will take to train the network with one data file, and for one training cycle which may be an important consideration in the near future.

I begin by using my network generator from figure 2 to create two networks. Each network is made 50% connected with 64 inputs, 1 output. One network is 64 nodes wide by 10 nodes deep, and the other is 128 nodes wide by 5 nodes deep. I have noticed a problem with the network generator. The final layer of nodes must be fully connected to the previous layer, otherwise a great deal of the network is useless, as it will never impact the outputs. I need to modify the network generator code to fix this problem. I simply modify the condition to connect a source node to a destination node by including the case where the node number is greater than the number of hidden nodes + input nodes:

```
if     ((randval<=connectivity)||(source>(inputlength+hiddenlength*(row-
1))-1)||(node>(hidden_nodes+inputlength)))
```

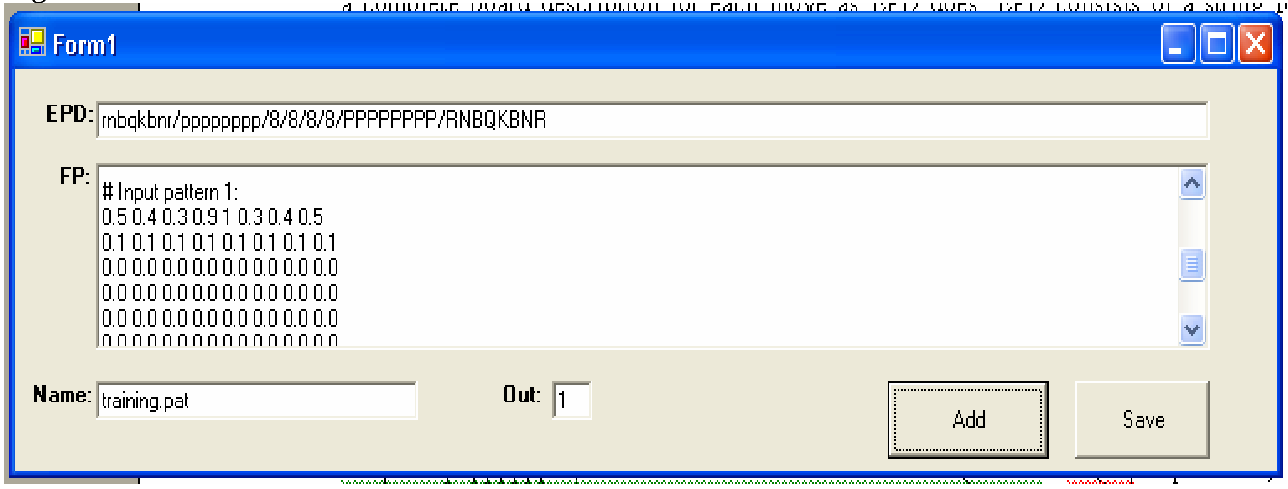
The new code may be seen in its entirety as NetGen3 in the journal files folder. Now all nodes in the network should be ensured to impact the output in some way.

I generate network5x128_041028.net and network10x64_041028.net which may be viewed in the journal files folder.

The goal now is to use the same set of input vectors to train both networks in order to see which network (with equal number of nodes) trains faster: the wide, shallow networks or the narrow, deep network. Connectivity is 50% in both case, and node count is equal. The only variable factor is the dimensions. Although the networks to be used in the real training will be much larger than these, this experiment will offer some insight into how the should be designed. More nodes will allow more training samples will be memorized. However too many nodes may lead to memorization and not schema recognition and generalization, which is obviously not desired. Therefore some middle ground will be sought. The number of layers should have some relation to the degree of non-linearity the network is able to "estimate," but Dr. Malinowski feels 3 or 4 layers is the maximum that would be useful in this respect. However, more layers will still "learn" so they are not totally useless. Making the middle (hidden) layers wider could lead to more relationship development (we allow more combinations of input data to be assembled). I would predict the wider network will also train more quickly.

I need to create a training data set. At this point I need to decide if making up random data would be the best solution, or if I should produce a simple program to convert existing EPD files into floating point values. I decide to create the program as I will need this functionality at some point when creating the data processing programs anyway. This program steps through the EPD string one character at a time and appends the floating point results to the end of a rich text box. The file may be saved as a .pat file for use in SNNS. Figure 9 shows a screenshot of this program. The program is called EPD_FP and complete source code is in EPD_FP_041028.cs, in the journal files folder. I create the training file which can be seen in training_041028.pat in the journal files folder by using 5 of the EPD strings in figure 6 as samples.

Figure 9: EPD to FP Data Generator screenshot



EPD: rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR

FP:
```
# Input pattern 1:
0.5 0.4 0.3 0.9 1 0.3 0.4 0.5
0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

Name: training.pat

Out: 1

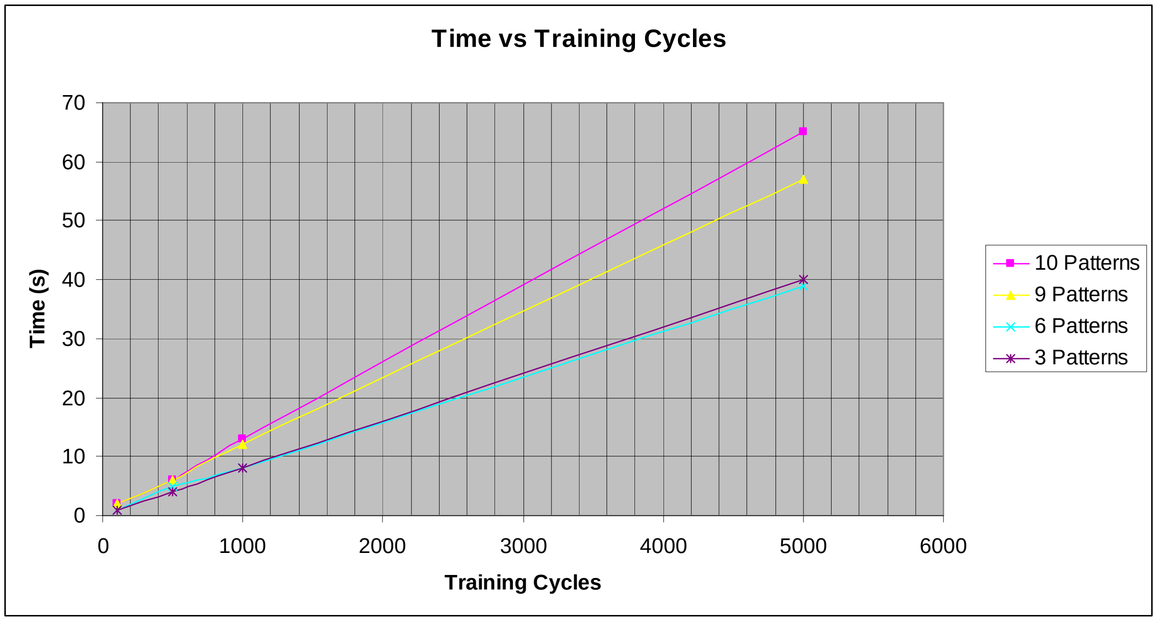
The training file just created has 5 entries in it. I will begin by opening the 5x128 network and training it with the data set.

I set the training mode to 100 cycles, 1 step. Learning constant is .2 and dmax is .1. These values will be kept the same for the rest of the day unless otherwise noted. The training process takes only 3 seconds (with the graphics window closed). I am surprised how fast the training is, and was expecting it to take much longer. This result is very promising…although this data is obviously highly simplified. I now do the same for the 10x64 network. There is no noticeable change in the learning speed, although I definitely would have expected to see one between the two networks tested. It seems that learning time is under 1 second per position when a file is run through 100 cycles. I keep the 10x64 network open and try 1000 cycles, step size 1. This takes 7 seconds to complete. 5000 cycles? 33 seconds. I decide to time the training for pattern sets of various sizes. A new pattern set is created, called training2_041028.pat. It contains 10 patterns, with outputs 1 or -1 (instead of 1 and 0 used in version 1). I will use SNNS to train 10 patterns and down and record the time needed for 100, 500, 1000 and 5000 cycles. The results for the trials are shown in figure 10.

Figure 10: Time (Seconds) needed for training in SNNS

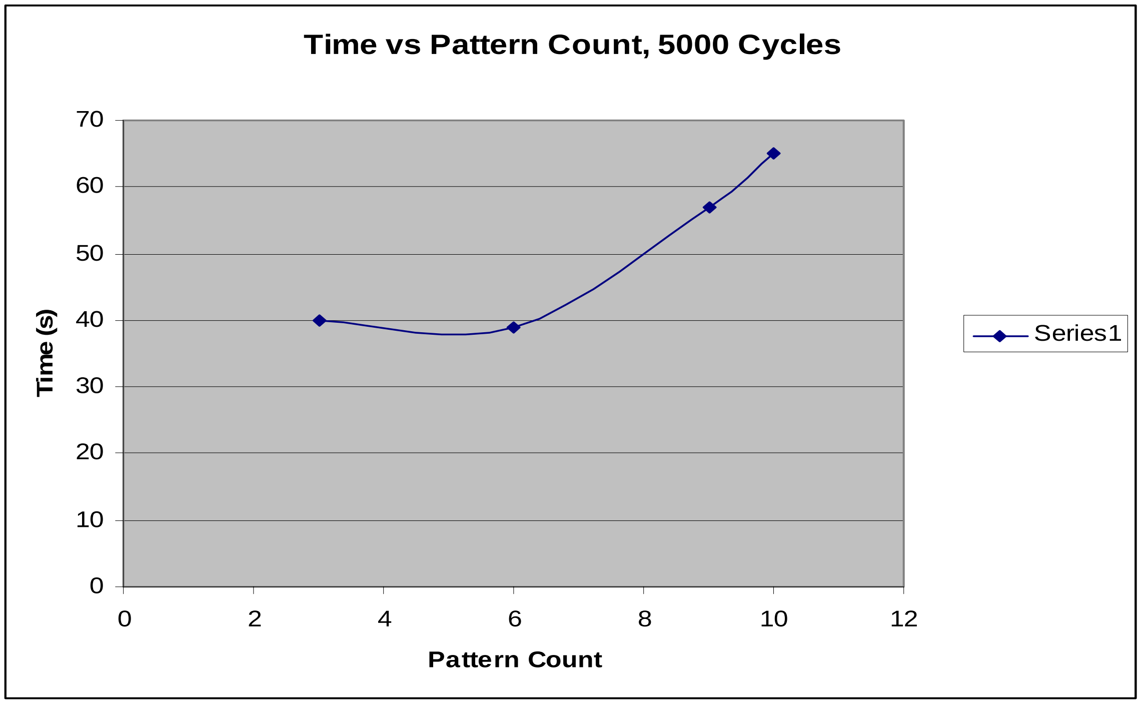
Patterns \ Cycles	100	500	1000	5000
10	2	6	13	65
9	2	6	12	57
6	1	5	8	39
3	1	4	8	40

Figure 11: Plotted Data from Figure 10



From figure 11, it seems that there is obviously a linear relationship between the number of training cycles and the time needed to complete them. There is no surprise here. But what about between the size of the training set and the time needed for a constant number of training cycles? Figure 12 shows the time needed to train 5000 cycles of various pattern file sizes.

Figure 12: Time needed to train 5000 cycles of various pattern file sizes



It seems that the pattern file size has a more non-linear impact on the training time required. Although only 10 patterns were tested as a max, it seems obvious from this plot that training time is going to be minimized by keeping a fairly small quantity of patterns in the training files. It may make sense to create a batch file to do the training, which will cycle through the data files. How long is training estimated to take?

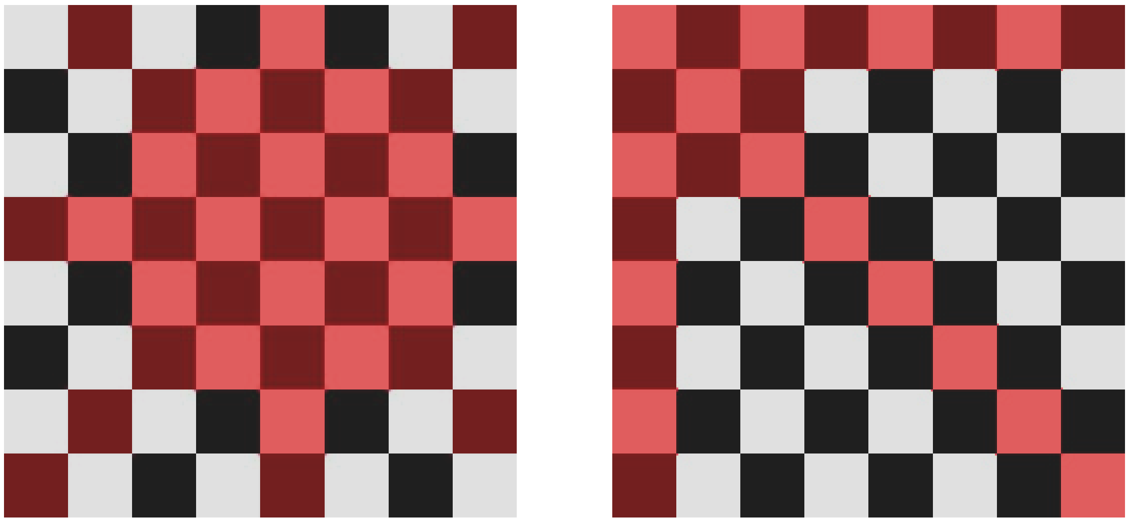
For one network (geographical approach as described in the functional description), I will make the following assumptions:

I have about 4 million games to work with. Half will be won by black and usable for training. I assume each game will have perhaps 40 positions…based on:

“Chess is a fascinating game to both play and study from a psychological perspective. Its complexity assures that the game will never be completely solved, like tic-tac-toe. Given an average of 30 possible moves per turn, and an average game length of 40 moves (80 half-moves), we can see that the game tree is at least 30^{80} nodes big (on the order of 10^{120}).” (Source: Mark Jeays). See A brief survey of psychological studies of chess.htm in sources folder. Original URL: http://jeays.net/files/psychchess.htm.

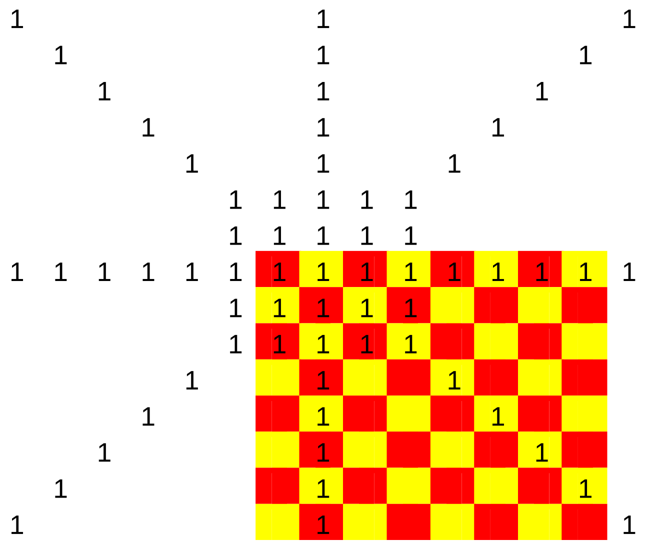
Thus, 80,000,000 individual board positions should be trainable for each network. Using a default of 100 cycles for a single training session, I would predict about 2 seconds needed for every 10 positions learned based on figure 10. Therefore, approximately 4400 hours would be needed for training each network with all positions! Obviously this is not going to take place. A sort routine will take place first, which will consider all board positions, and categorize them based on the move black makes. Of course, in order to make this calculation the total number of legal moves must be determined. This is *M* as described in the functional description. To determine *M* I will consider a chess piece as though it is made of a queen plus a knight, which would cover every possible move in the game at any time. Now, the board is considered empty other than this piece. Therefore, if this special piece is moved to all 64 spaces, we can record the total number of moves which are possible. The number of highlighted squares (figure 13) is simply summed for all 64 squares to get a total. Notice that figure 13 shows the top left corner position and one of the 4 center positions under consideration.

Figure 13: Determining the total number of legal moves



I decide to do the calculation in Excel. This is shown in figure 14.

Figure 14: Excel move calculations…



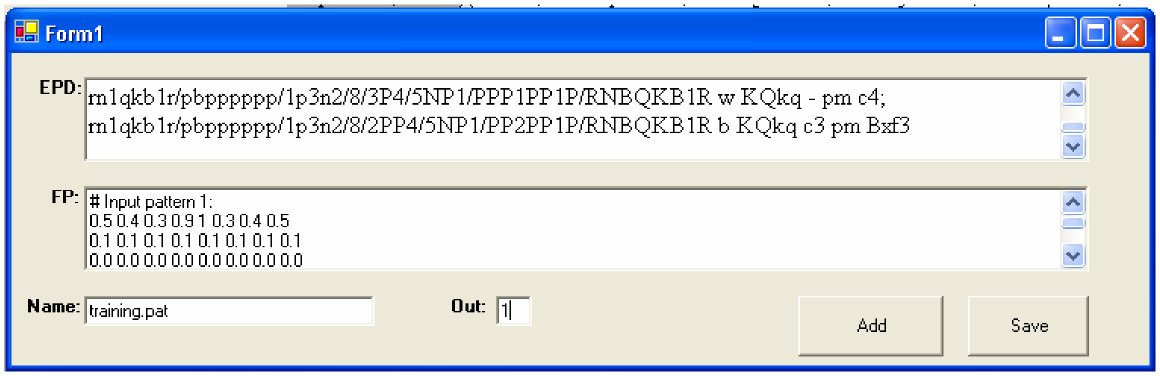
24	25	26	26
25	28	30	30
26	30	34	34
26	30	34	36
26	30	34	36
26	30	34	34
25	28	30	30
24	25	26	26

1856

There are 1856 possible moves to be made at any given time. Thus, if I consider that 80,000,000 board positions exist in my training set, about 43100 positions would go into each category, taking roughly 2.5 hours to train. This is only for the “yes” decisions. An equal number of “no” cases would also have to be trained, meaning about 5 hours would be needed to train each network (some will be more or less, as not all moves will have equal complexity). One PC could train about 4 networks per day in a best case, which means 100 PCs would take about 4.5 days to train everything. This is possible, but still daunting. I would like to somehow reduce the problem to take 20 PCs 4 days to train. This could be accomplished in one lab, and this block of time could realistically be reserved over weekends, etc…

Although figure 11 seems to give a clear linear relationship between training time and training cycles, as expected, the results shown in figure 12 were unexpected. It seems that this relationship should have been linear, and perhaps it will appear as such if larger training datasets were considered. I will improve the EPD_FP program to accept an entire file of EPD strings, rather than just one at a time like it does now. I simply add an external loop to the current string parser, which will go through lines of EPD strings one by one. The new code may be viewed as EPD_FP2_041028.cs in the journal files folder. The GUI is slightly modified as shown in figure 15.

Figure 15: Modified EPD_FP program interface to accept multiple EPD strings



EPD:
```
rn1qkb1r/pbpppppp/1p3n2/8/3P4/5NP1/PPP1PP1P/RNBQKB1R w KQkq - pm c4;
rn1qkb1r/pbpppppp/1p3n2/8/2PP4/5NP1/PP2PP1P/RNBQKB1R b KQkq c3 pm Bxf3
```

FP:
```
# Input pattern 1:
0.5 0.4 0.3 0.9 1 0.3 0.4 0.5
0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

Name: training.pat

Out: 1