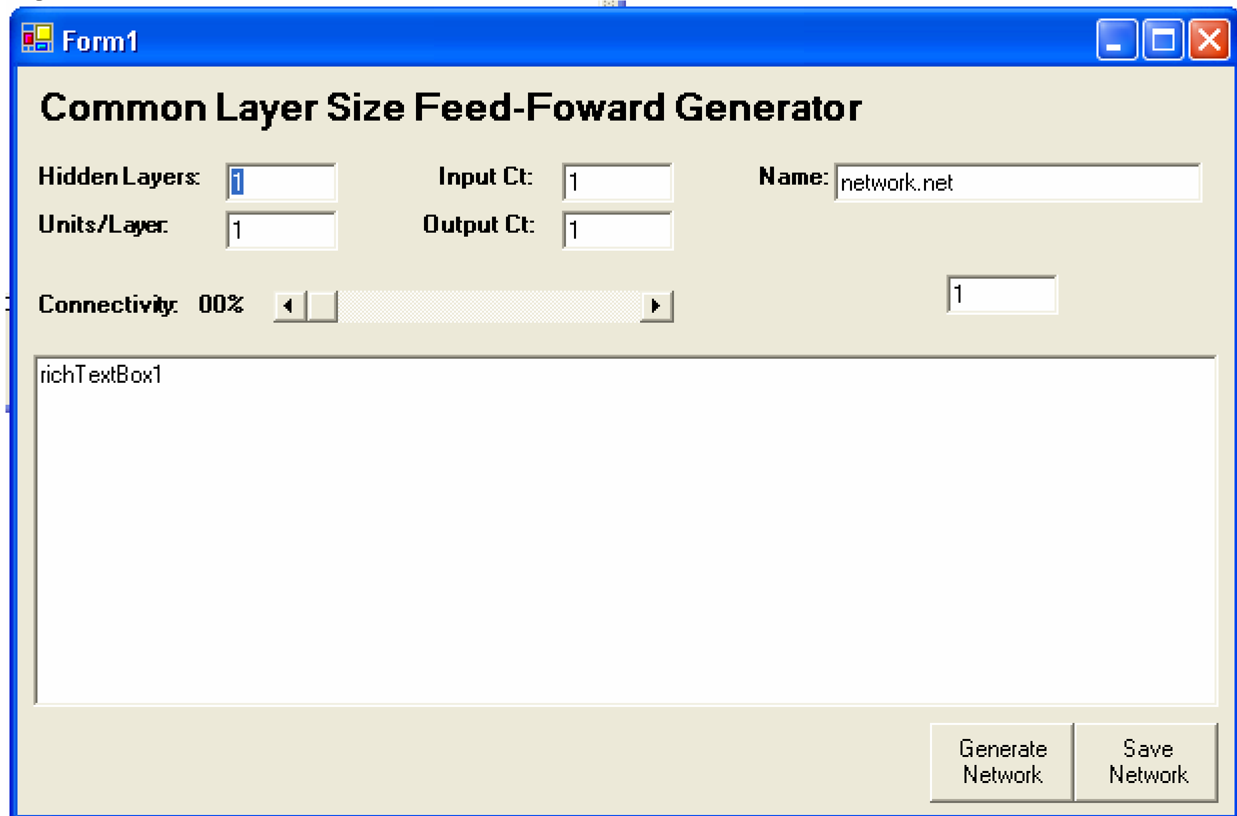
10-17-04

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

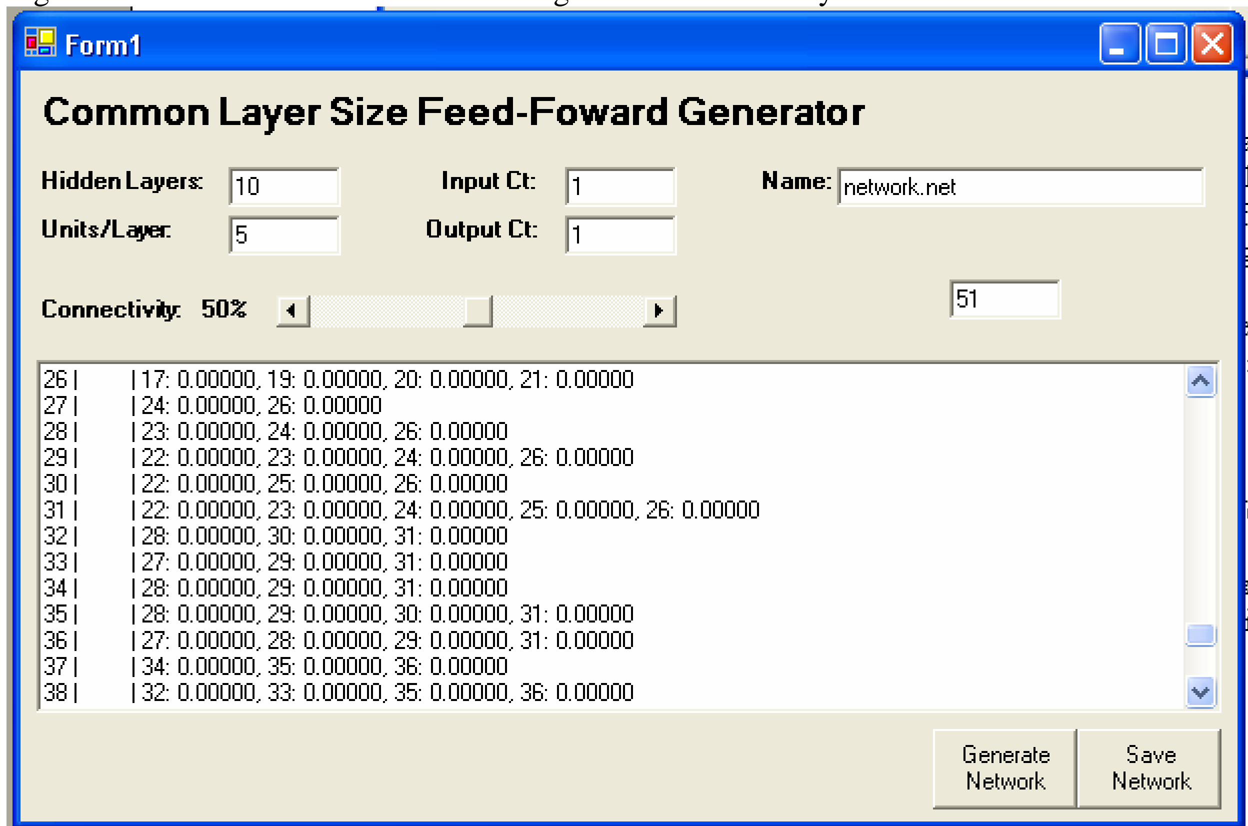


Debugging has gone fairly smoothly, the main issue being that having a network with layers more than about 50 nodes tends to become extremely slow. The problem was using the append member of the richTextBox class to add to the output file. I have found it is better to create a string for each new line 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.

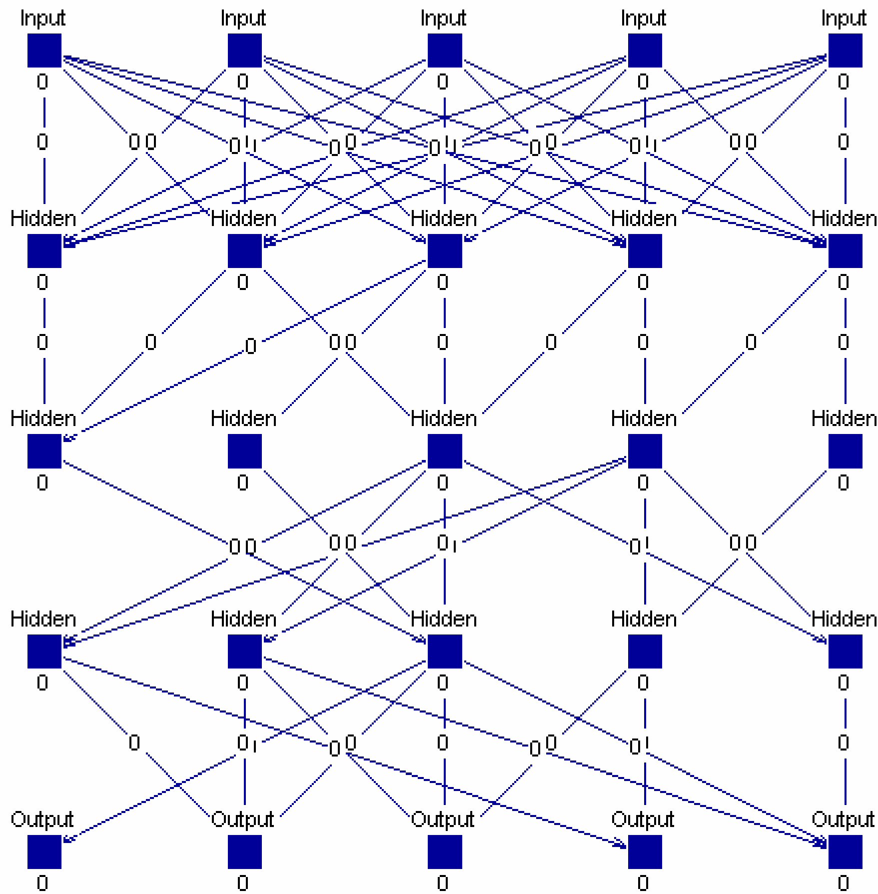
Figure 2: New Network Generator Showing Partial Connectivity Network



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

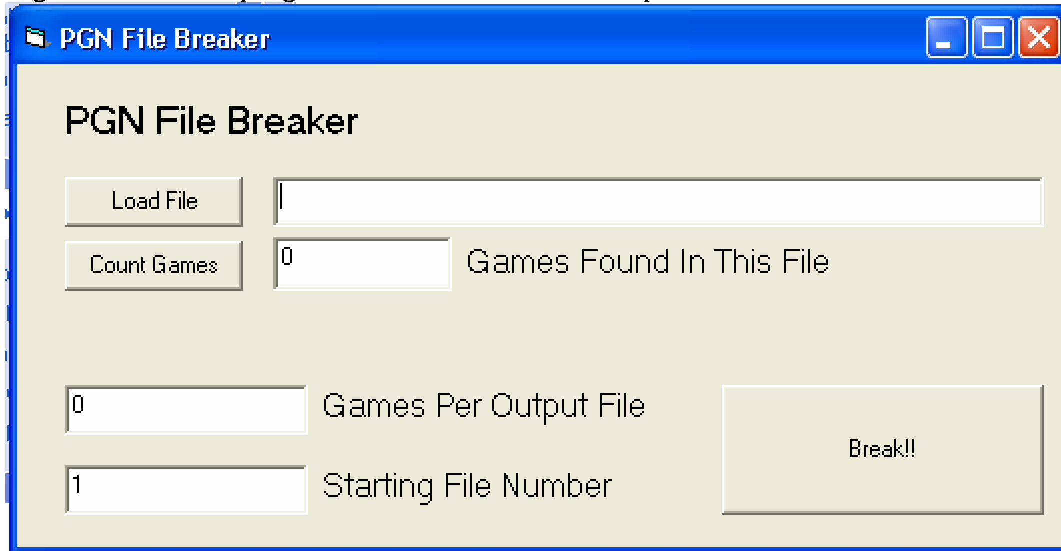


With the network generator now fully functional, it is time to start considering the data file processing. The dataset I will be using is capable of producing PGN (portable game notation) files, which are algebraic, standardized Chess game recordings. ChessBase 9.0 (NEED TO ORDER!!) should be able to 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



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

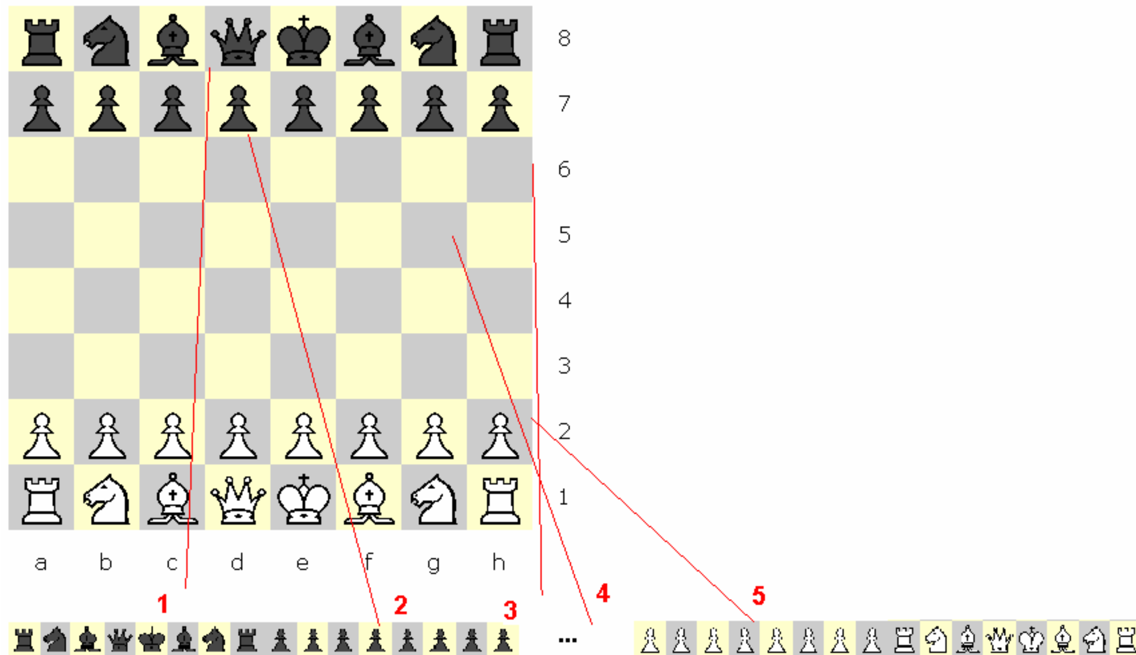
a complete board description for each move as EPD does. EPD consists of a string for each move in the game, a typical example of which is shown in figure 6.

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?