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
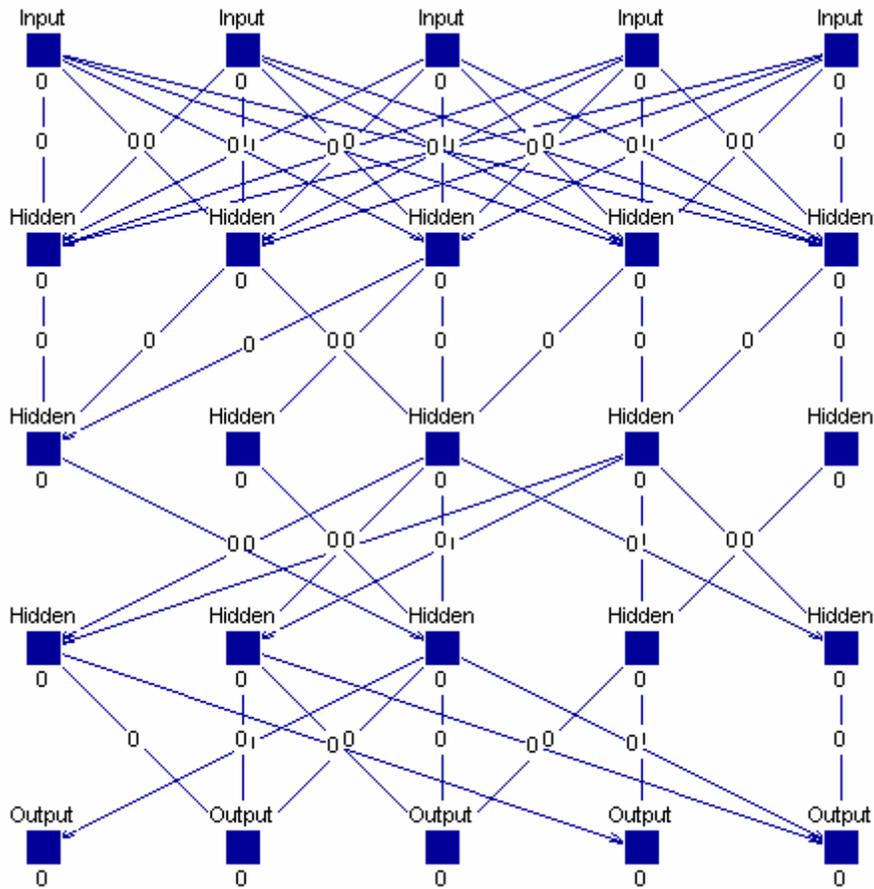
With the network generator now fully functional, it is time to start considering the data file processing. The dataset I will be using is capable of producing PGN (portable game notation) files, which are algebraic, standardized Chess game recordings. ChessBase 9.0 (NEED TO ORDER!!) should be able to 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
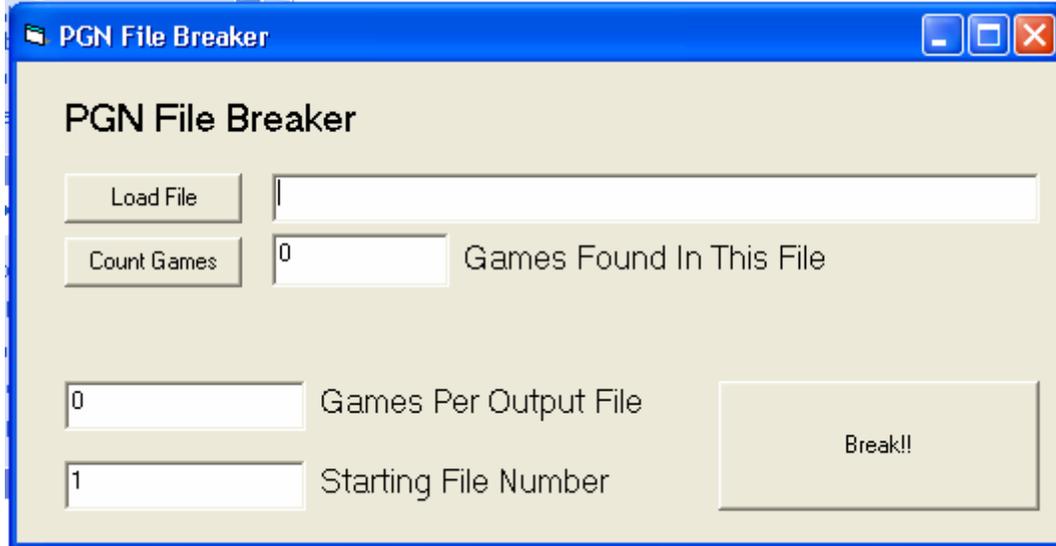
The PGN Breaker program is written in Visual Basic 6.0, and is simply a text parser. The code is shown in the Journal Files Folder as "BreakerCode." An example of the PGN format is shown in figure 5.

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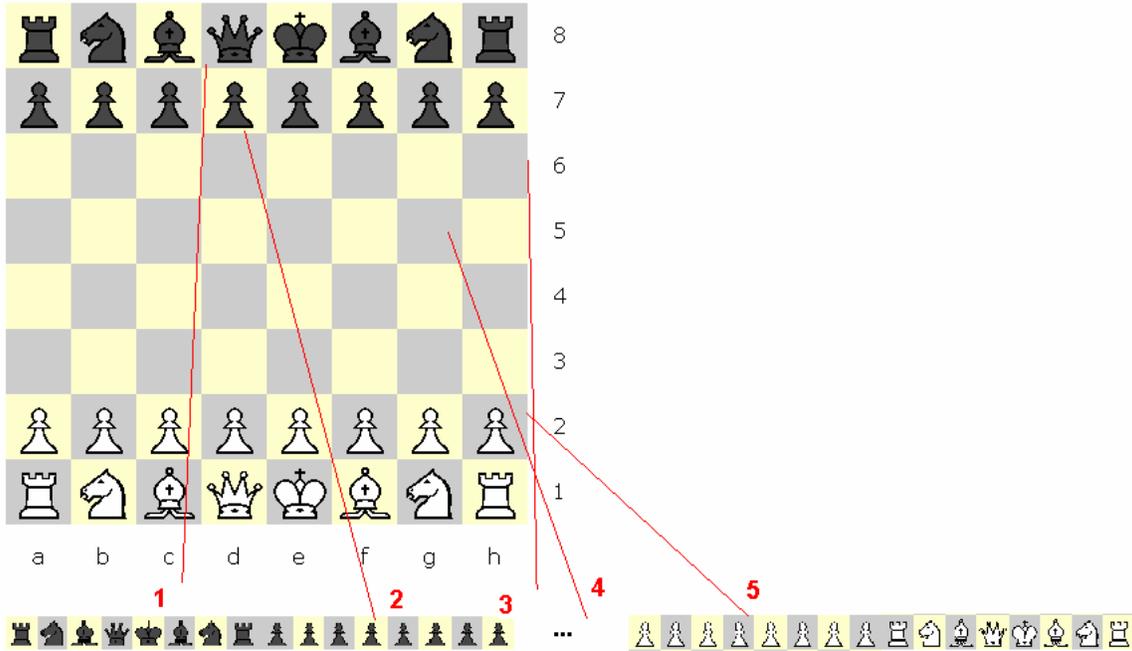
The inputs into the neural network will be in the same order as the positions are arranged for EPD format. Because the inputs to the network must be floating point values between

+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?

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.

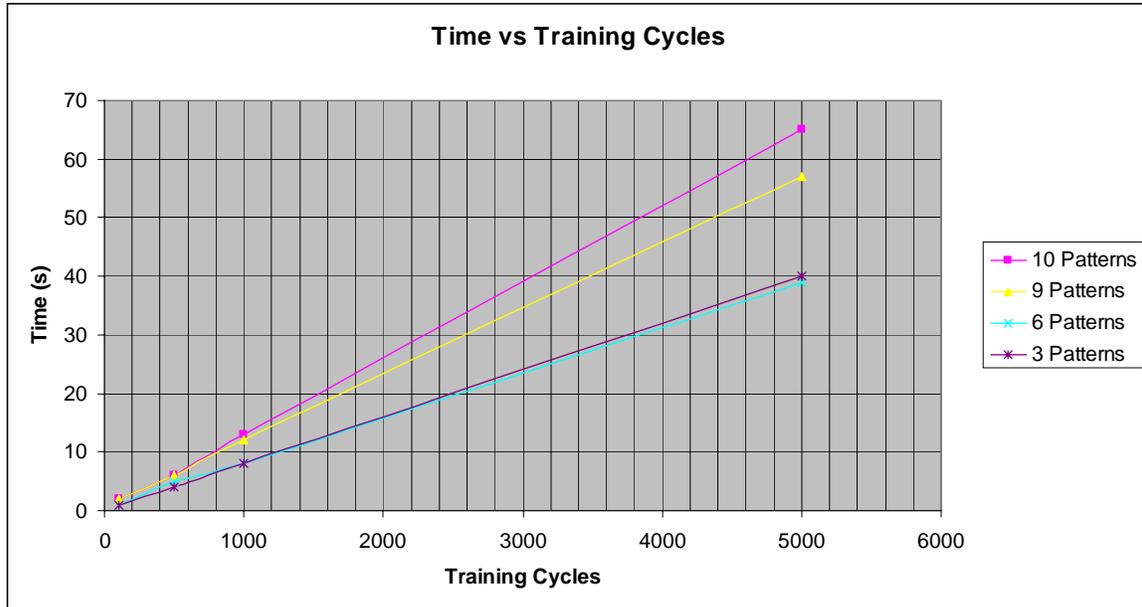I generate network5x128_041028.net and network10x64_041028.net which may be viewed in the journal files folder.

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

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

Time vs Pattern Count, 5000 Cycles
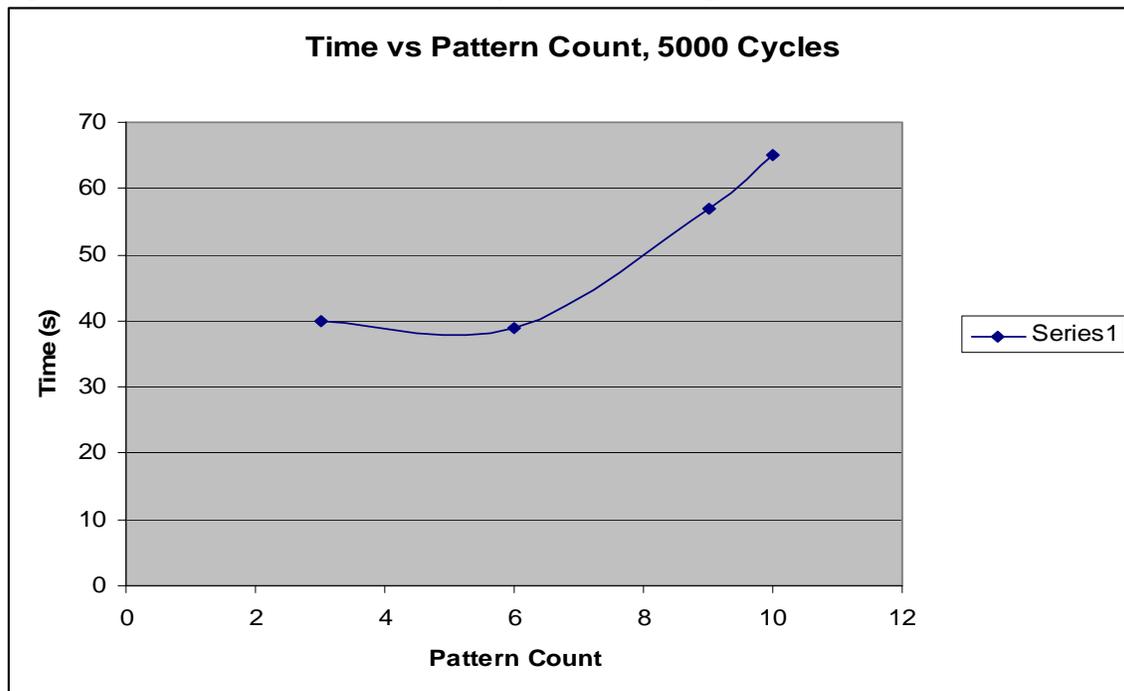
It seems that the pattern file size has a more non-linear impact on the training time required. Although only 10 patterns were tested as a max, it seems obvious from this plot that training time is going to be minimized by keeping a fairly small quantity of patterns in the training files. It may make 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

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

10-30-04

I am curious to test the program in figure 15, so I create another training file, which ends

up having 137 games in it. The file may be seen as training137_041030.pat in the journal files folder. I load the 10x64 network in SNNS and select the above file as the training pattern set. I am simply curious to see how long 100 training cycles takes, and find 18

seconds are needed to complete the task. This is better than I was expecting, as previously 10 samples needed 2 seconds. I will work on creating sets of larger pattern counts shortly to see if I can obtain a linear result for the training times.

Currently, a more pressing matter is the Argonne Symposium where I will be presenting this project and the current status. I will be presenting the slides found in "Argonne_041030" in the journal files folder.

I create training files for 137, 332, 432, 730, 1127 and 1127 patterns which I will now use to train the 10x64 network. The training times (in seconds) are shown in figure 16. 100 cycles, step size 1 is used for all training. The network is always re-initialized between sets, and all learning parameters are kept at the default settings for this test.

Figure 16: Training time data

Patterns	Time (s)
137	18
332	44
432	65
730	130
1127	219
1227	234

Figure 17: Plot of figure 16, showing nearly linear relationship
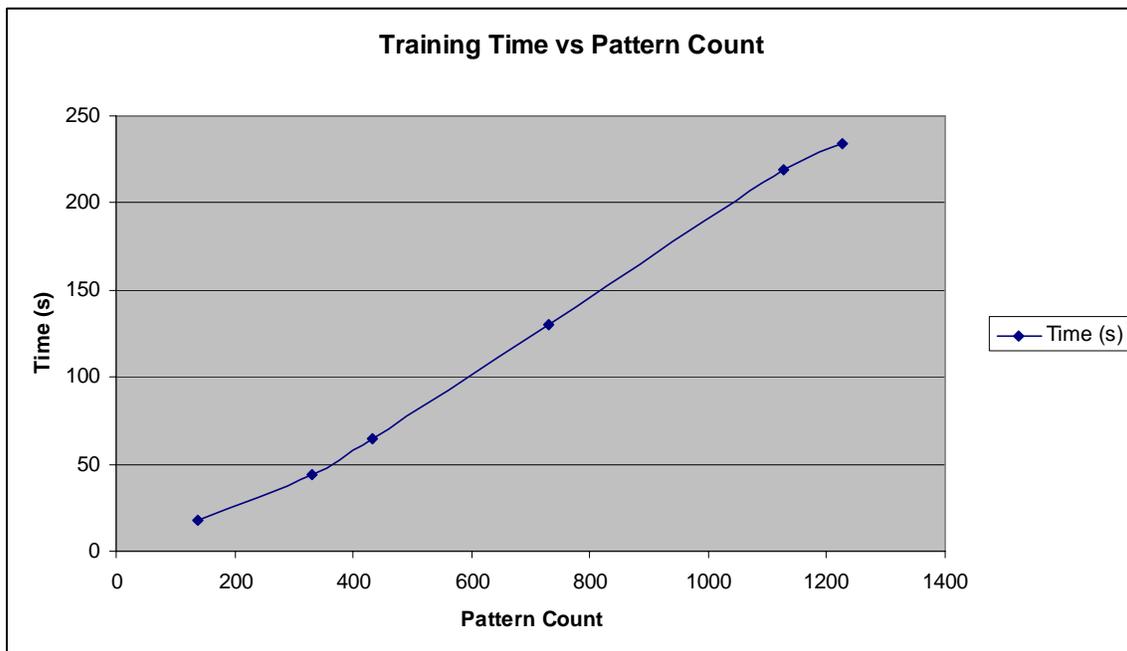
After completing the training, it appears that training time is actually a linear function of the pattern count, which was what the assumption wad to begin with. The discrepancy seen with lower pattern count is likely due to loading time, reaction time in starting and

stopping the timer, etc. According to the latest results, it makes more sense to keep a fairly large number of patterns in the pattern sets in order to reduce the time used in loading and switching sets. To come up with the equation for this line, the intercept is forced to 0 in Excel as training 0 patterns must take zero time. The equation is simply:

Time(s) = .186 (Pattern Count)

This equation will be used to approximate the set size needed when training the networks for a specified amount of time, which will likely turn out to be the most practical approach to the problem. A plot showing the trend line and the R^2 value is shown in figure 17.

Figure 17: Figure 15+trend line and R^2 value
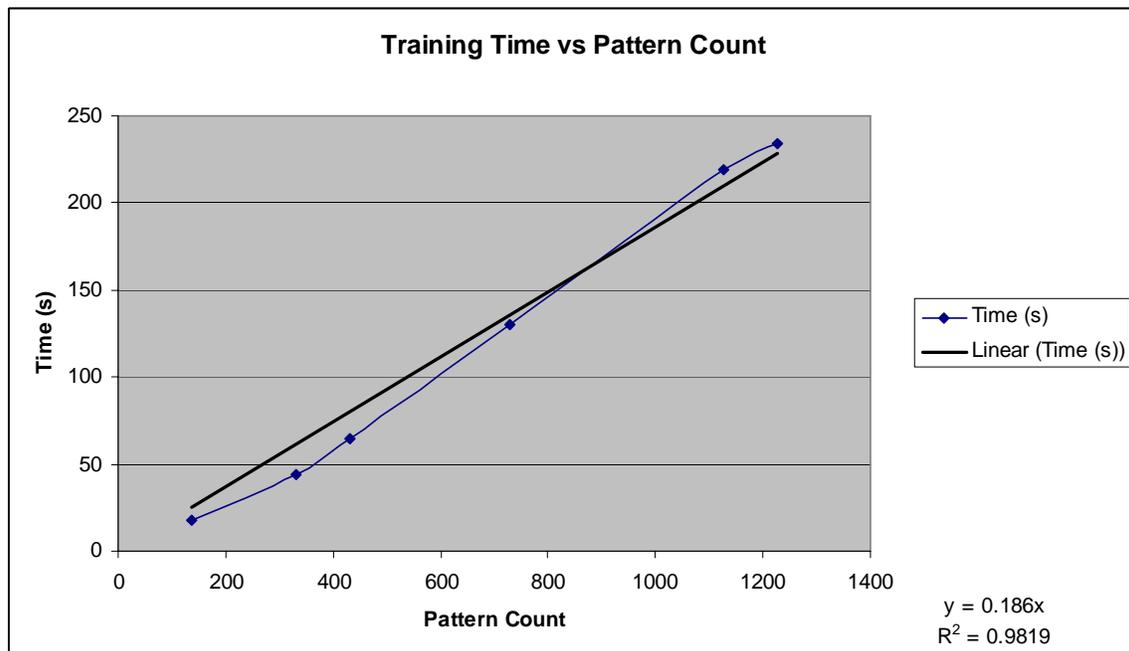
The R^2 value is close enough to 1 to have a large degree of confidence in the equation over the range of this data. Thus, each pattern will be considered to need .186 seconds to train.

Training time is based on running Java SNNS on a 3.06 GHz P4 (Northwood core) with hyperthreading and 1GB of DDR RAM. Hard disk is 5400 RPM and FSB runs at 533MHz. Training times are expected to vary considerably when SNNS is run on different machines!

11-4-04

I have spent some time making final changes to the Argonne presentation and also adding two new slides describing the mathematics behind the functional and geographical design

approaches which were outlined in detail in the functional description document. The final version of the presentation is saved as "Argone_041104" in the journal files folder.

Yesterday, ChessBase 9.0 arrived. I installed in last night, along with the Mega database and the Corr database. So far, it seems the database program works as advertised, and will be suitable for creating data sets for the network modules to use in the learning stage. The search function allows a specific move (maneuver) to be specified and it will return all games containing the move in the database. Most likely, this feature will be the most valuable in creating the individual datasets.

Today I will spend time working with ChessBase in order to understand the abilities of the program and to come up with the best possible way to extract the data I need for training. I hope to come up with a complete plan and also start the data gathering process by the end of the day today.

I also want to keep in mind the following: ChessBase 9.0 ships with an endgame database on 5 DVDs which contains all endgames for 6 or less pieces on the board. Thus, it is possible to play these positions 100% perfectly by looking in the database. Do I want to pursue integrating this database (which is going to substantially improve performance) or do I only want to base endgame performance on the saved game data. It is not possible to "extract" endgame positions and train them as the other games, so these seem to be the only two answers to this question.

I begin by creating a new database, and copy the contents of the Mega database, Corr database, and the games I gathered this summer (about 100,000) into the new database, called FULLDATA. The games I gathered are in PGN format, and were downloaded from the internet at dozens of sites offering saved chess games. Now that a complete database of ALL data exists in one place, I should be able to process it more quickly and not have to jump between numerous databases. The copying process takes about 45 minutes in all, which is less than I though it would. In all, the complete experimental database now has 3,202,623 games stored, which I hope is enough to learn the game! I now will perform some maintenance on this database:

I begin by removing all games in which the light side wins. Because I will be training the dark side, I want data in which a win or a draw occurs, which would mean that dark had a "winning" strategy developed throughout the game. It would be nearly impossible to go through all of the 1-0 games (white wins) and find the "bad" move for black (which would allow the rest of the game to be used in training). Thus, it is best to cut this data (entire games) out of the training set. Doing so does not mean that "bad moves" (or non optimal moves) will not exist in the data, as they certainly will exist, but it simply means that such moves did not result in a loss to white and therefore COULD still be considered a "proper" move to make when considering the board position at that time.

Operating with this large database is very time consuming…in fact deleting the games in this fashion is not practical at all as it will take over 12 hours to complete…(By sorting by result and then deleting)…

Due to some concern that the games collected over the summer do not confirm to the same standard as those in ChessBase, I decide not to use them. I create a new DataBase, this time with the games from the Mega database and the Corr database. It has as total of 3110269 games. I backup this database. Now, I search for games in which white wins (1-0 result). Now I delete these games. Once this is done, I remove them from the database, which only takes about 20 minutes to complete! Obviously, this is the way to remove games in the future! Now I have a complete set of 1978263 games in which black wins or draws. Further game removal is not required. Now, I decide to come up with a way to actually get the datasets required for training. Each network will need its own dataset, composed of both "yes" decisions and "no" decisions (the geographical, move based approach is being considered first). ChessBase has a nice search function which will find the specific moves requested, but it does require that a piece be specified.

I find out at this time that the database also contains a couple hundred text files of tournament listings (results, etc.) which are "in the way" of the real data, so I decide to remove them from the database just created. I delete them and repack the database again. 1902248 games are left after the latest repack.

I now will need to remove the annotations from the database, backup the database, and finally get the datasets required for training. The de-annotation process is successful, and I now make another backup of the final database. The current games are now ready for "sorting" or classification by move.

I make copies of a chessboard on paper in order to keep track of the move sets I have saved. Figure 18 shows a screenshot of how the search is configured.

Figure 18: ChessBase 9.0 search configuration (for moves)
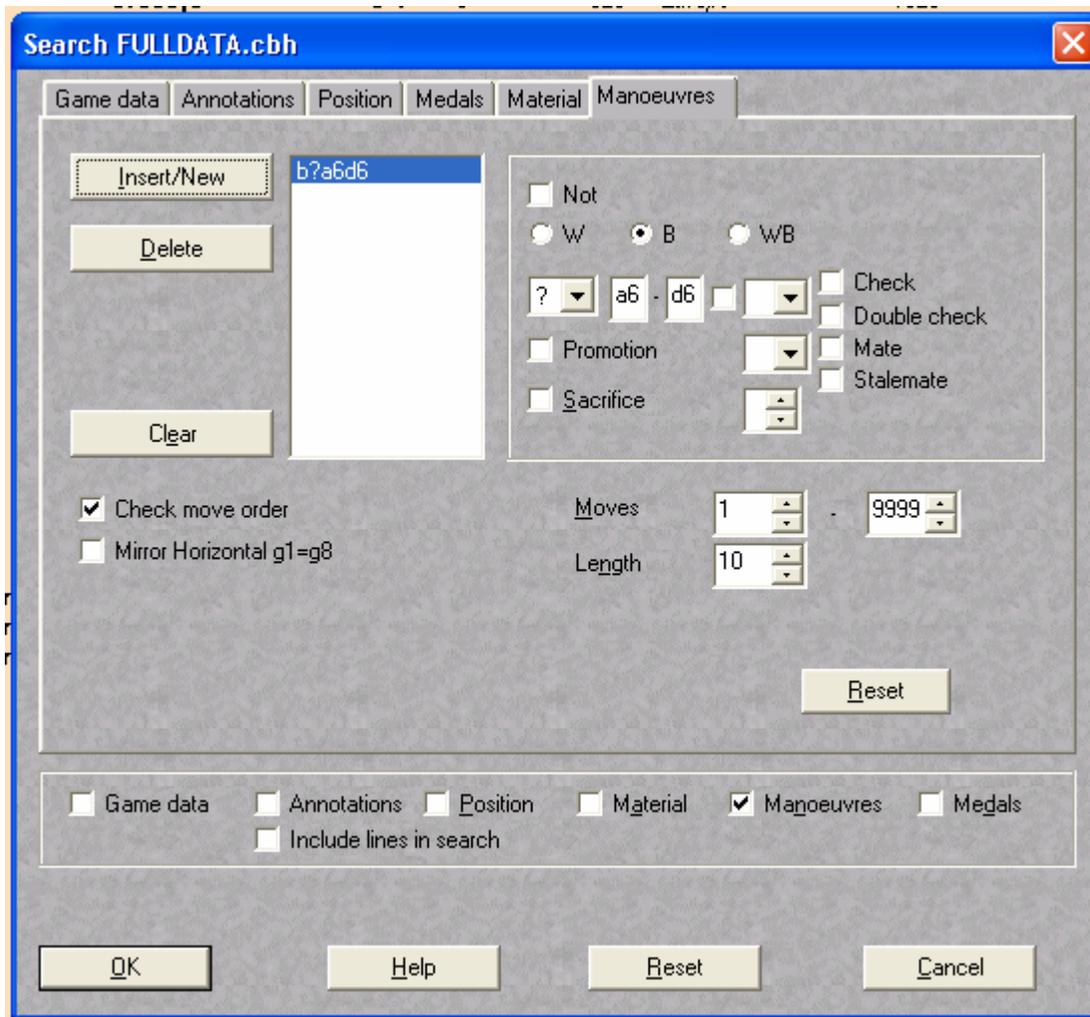
This same search will be performed for each and every move in chess, the total number which was calculated earlier to be 1856. After performing numerous saves, I find it takes roughly 1 minute for each move. About 30 hours (of manual searching) will be needed to collect all of the needed data. I install the database on an older P3 1.0 GHz system so I can have two searches running at once. I am going to try to have all data collected by next week, 4-11-04. Backups of all data will be put on DVDs, and then the remaining preprocessing stages will be carried out.

11-11-04

Currently, data has been extracted from the database for all moves which have initial positions A1,A2,A3,A4,A5,A6,A7,A8,B1,B2, and B3. The process has been taking much longer than anticipated. The 2nd computer being used in processing failed on November 10, so it is no longer being used in data extraction.

Fortunately, a slightly improved technique can be used to speed up extraction on a single PC. ChessBase allows a wildcard search for one of the positions (initial or final). Therefore, it is possible to create databases which contain all moves from a specific

starting location. These databases are much smaller than the entire database, so searching them takes much less time. It is possible to extract an entire move in less than one minute from the smaller database. Figure 19 shows how the smaller databases are generated for each initial position.

Figure 19: Making smaller (initial position specific) game databases
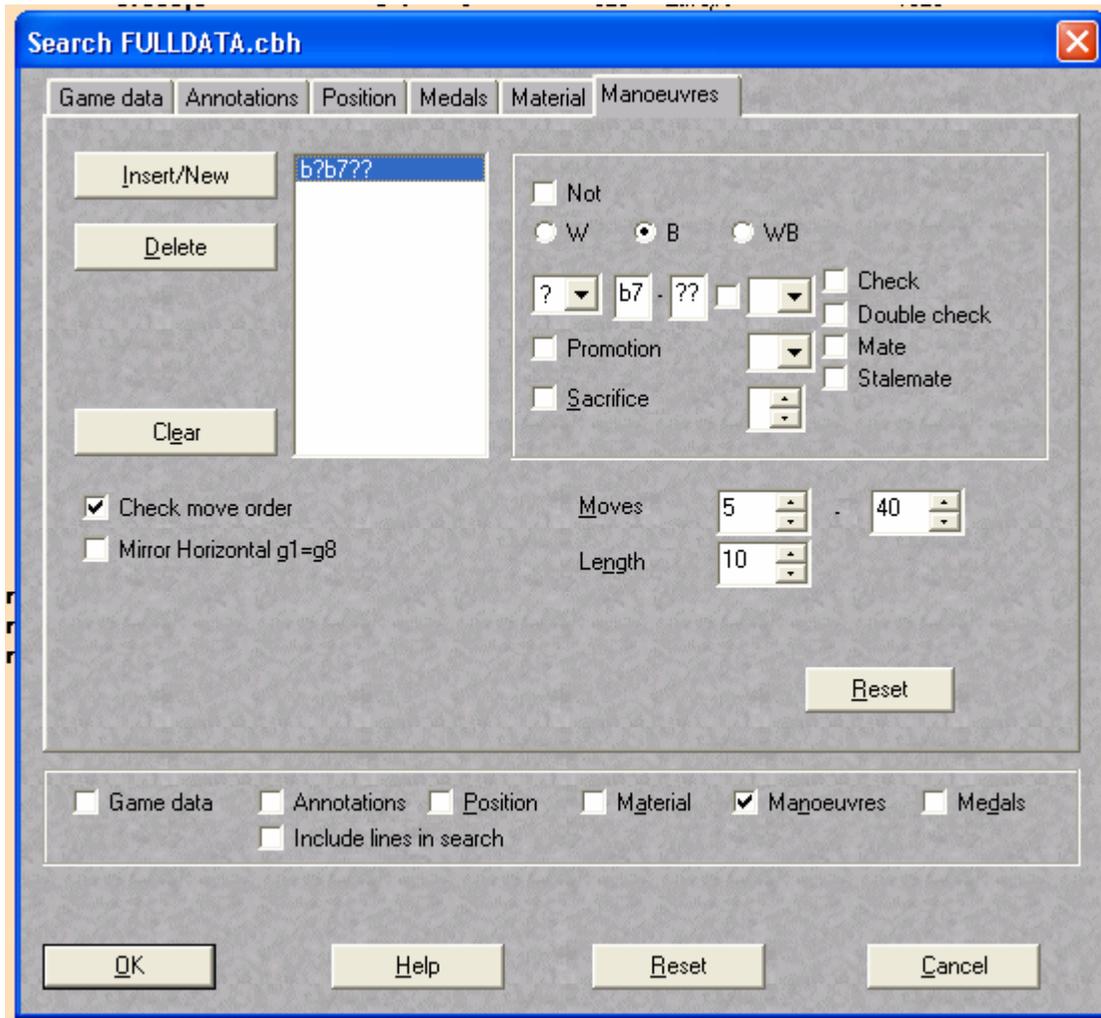
Today will be spent on data extraction, as it must be completed before any useful training can be performed. It may be possible to look at only one move, but evaluating this network as a standalone unit may be highly difficult, as we really need to evaluate performance over the entire game.

I am backing up the extracted data after every 6 initial positions are evaluated. Zipping the pgn files reduces the size by about 75%, which will save a great deal of space when sending the data to the Gdansk.bradley.edu server. The backup does however take some time to complete (45 minutes for 6 initial positions).

The laboratory directory has agreed to install Java on 4 machines in the lab which will be a start for my training processes. With any luck this stage of the project should be ready to proceed in under two weeks, as the lab director has also agreed to install the database on another machine in lab so that data extraction may once again enjoy increased efficiency.

I have some concern of the greatly varying game count which appears in the final game collections. Some popular moves (b7 to b6 for example) have over 100,000 games! Others, such as corner to opposite corner moves, have as few as 300. I expected from the beginning that different numbers of games would be found for each move, but I did not expect quite this much variation. I will need to come up with some idea on how to deal with this, as it may now require that the networks be made of varying size as well. The other possibility, which I currently prefer after having had some positive feedback at the Argonne Symposium, is to proceed as follows:

Create all networks the same size, and large enough to deal with the largest of the datasets (which is still based on estimation). Then, initialize all networks and only provide perhaps 10% connectivity (they are still to be feed forward networks of common layer size). The idea is that lower connectivity, as it trains faster, will provide the same end result as simply having fewer nodes. By observing the output error plot (example shown in figure 20), it is possible to determine when the maximum amount of training has taken place, as the error graph should begin to rise again after having reached some minimum value. In the cases of the datasets containing very large sample counts, I expect this will happen far before all samples have been trained. Training with more samples is expected to give a better network in the end, so I do not wish to simply stop "halfway through" the data, for example. Because the overly trained network only has 10% connectivity, I can easily add more connections through editing the .net file. New connections will be assigned weights of zero. In doing this, the new connections have absolutely no impact in the network at this point. However, the storage capacity of the network will have been increased, and new patterns should now be "learned" through manipulation of the old weights, but more importantly the newly added connections will be utilized as well. Thus, the process is to be repeated until a given dataset has been adequately trained.

Figure 20: Example of error graph to determine occurrence of "overtraining" in networks
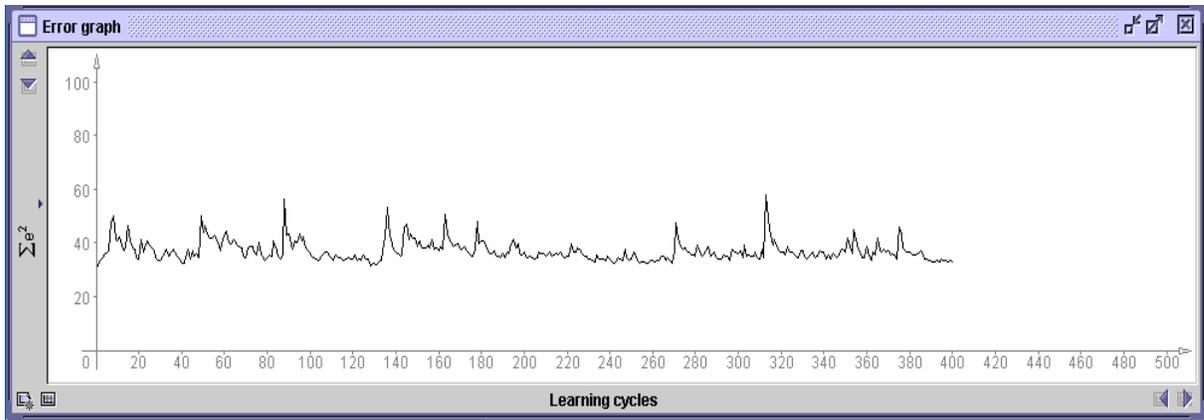
11-18-04

This week Dr. Malinowski and I have considered the possibility of employing radial basis function networks (RBFNs) instead of the current approach which is using hyperbolic tangent activation functions. The RBFN approach differs in three major ways from the current approach.

-The activation function is typically the Gaussian distribution function or some similar function, resembling a band pass filter as shown in figure 21.

Figure 21: Radial basis function network Gaussian activation function (Source: Mathworks: Introduction to…located in "sources" folder).
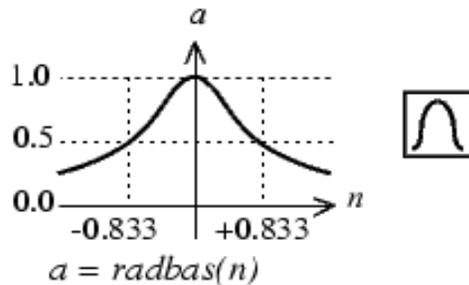
Radial Basis Function

-Each node creates an output by determining the "distance" (dot product) between the input vector and the weight vector…this "distance" is passed as the argument into the activation function, which means the maximum output exists when the two vectors are at "right angles" to each other—producing a dot product of zero—in a geographical point of view (the activation function is centered and maximum at zero). This is described in more
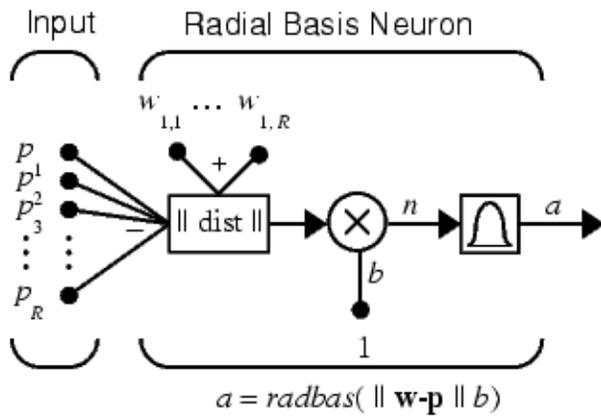
detail in   Introduction Radial Basis Networks (Neural Network Toolbox).htm   (located in sources folder).

The structure of the RBFN node is shown in figure 22.

Figure 22: Radial basis function network node structure (Source: Mathworks: Introduction to…located in "sources" folder).

$$a = radbas(\ \|\ \mathbf{w\text{-}p}\ \|\ b)$$

-The network will only have 1 hidden layer of nonlinear elements. An input layer and a linear output layer complete the network. Thus, a simple form of this network as represented in SNNS is depicted in figure 23.

Figure 23: SNNS representation of a simple radial basis function network
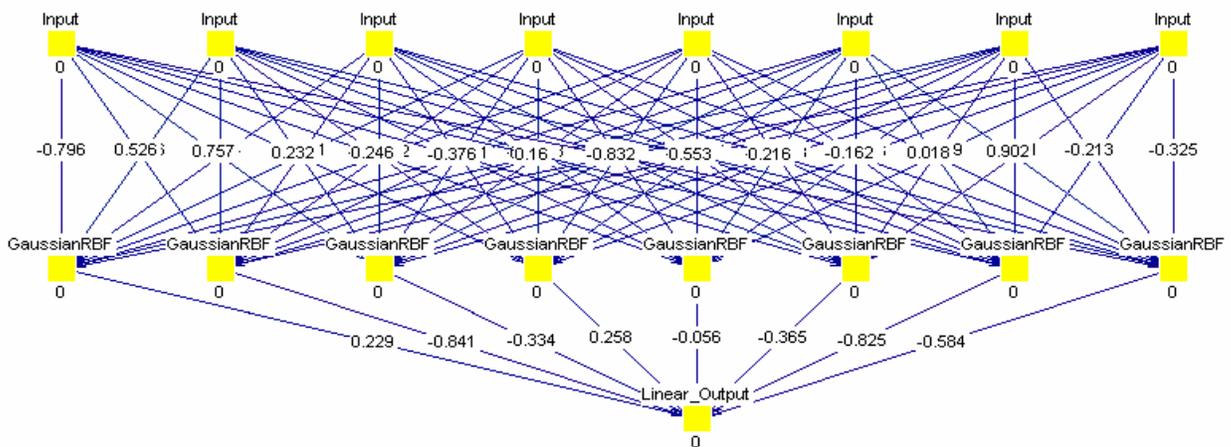
RBFNs are particularly promising for this project as they **are particularly good at pattern classification problems** if the problem can benefit from the fact that they make LOCAL APPROXIMATIONS or LOCAL CLASSIFICATIONS in each node (this is somewhat intuitive by the activation function's resemblance to a band pass filter rather than a low or high pass characteristic seen in typical feed forward activation functions). They also perform function approximation. SNNS is able to work with RBFN networks, so it should be possible to compare the performance of this design approach to that of the original approach.

There are a few concerns with RBFNs. The problem in question has 64 elements present in the training vectors, but there are an essentially infinite number of possible board patterns. The strength of RBFN networks is that they can determine if an input vector belongs to a specific pattern category (by measuring the distance from the input vector to the weight vector). My fear is that the number of nodes required to adequately match unknown board patterns to known patterns will be impractical (but this may also be the case with the current approach). It also seems that RBFN networks only work when the problem is governed by a continuous function: "It has been shown that, given a **sufficient number of hidden neurons**, GRNNs [an RBFN] can approximate a **continuous function** to an arbitrary accuracy." (Mathworks). Does this problem satisfy the criteria? ~~I doubt this at this point, which may mean I want to look into probabilistic neural networks…which are a subset of RBNs. This will be examined in the near future~~.

I initially felt the answer was no…but I have now convinced myself that chess is continuous, at least until check-mate. Discontinuity within the game would mean certain board positions would exist where a move can not be made **legally**. The only time this happens in reality is when a check-mate occurs. Since no moves are defined for a board position with check-mate, a plot of (the practically impossible) "move as a function of board position" would end as soon as a check-mate occurred. In the worst case excluding check-mate, a player will have *a move,* even though it may not be considered a *good move*. This is not a discontinuity. Considering an individual network in the move based geographical approach, each board position must yield an answer of either yes or no. If the rules of chess are obeyed, a discontinuous decision output *will never occur* until check-mate occurs. **Even in this state, producing a "no" output is valid…and it would therefore not be a discontinuity** anymore (it is interesting that this approach leads to this conclusion of a local continuity despite the global discontinuity).

At the very least, RBFNs are worth considering further. More research is required in the next week to make the choice between pursuing RBFNs or "traditional" networks—although at this point the best approach is to probably perform an experiment in training one of the networks required in this project and comparing the performance of an RBFN implementation and the feed forward design. Performance rating at this point would have to concentrate on the memorization of the training data and the observed "error." Also important would be the observed training time.

In either case, data will be the same for training. Thus, I will use the remainder of the day extracting datasets. ChessBase 9.0 has been installed on a machine in the lab (job248g) so I will be able to work much faster in completing the task.

A note on "error"…most neural networks are evaluated based on some sort of mean square error or average error. Normally, being +.2 or -.2 from the desired output is an equal amount of "error." This may not be such a valid way to look at this problem. First of all, a single network output is essentially pointless without comparison to other networks. Training will be done with yes or no, so ideally we would always want +1 and -1 outputs. This is not going to happen, and we may see values anywhere in between. Is +0.7 wrong when we want to see +1.0? This can not be determined at this point, and a true evaluation of performance is not possible until all networks are trained and the outputs are compared and move choices made. The best method to use in evaluating data set memorization may be to write a simple program which will determine the number of correct SIGNS (after all, all + values are yes and all – values are no, which is all we desired to train). ~~It may also be desired to know how much each correct sign varies from the ideal value.??~~

This consideration of error leads to ~~two~~ other considerations:

-Do we really want to train the network with +1 and -1 only? Is there more meaning in defining some sort of "relative move strength"? Two problems exist here. First of all, formulas to calculate relative move strength accurately simply do not exist, even though some versions of them are used in commercially available chess programs. No matter how complex they are, they are always little more than artificial models of an impossibly complex system. They are not going to be correct all of the time. Of course, actually implementing this equation is not a trivial task either, as it would have to examine a full game leading up to a certain board position for every board position in the training set...that is if an equation could even be derived in the first place since many of the better ones are proprietary information.

Most importantly however, this function would defeat the point of this project all along by adding some sort of external "expert knowledge" when the original goal was to use ANNs alone. Because of this fact alone, I will not pursue this further. However, I will leave open the possibility of adding additional network inputs at a future time so long as they are clearly visible on the board or in the game. Perhaps piece proximity data, last move made, etc.

12-1-04

A schedule is produced which may or may not be followed exactly, but it is a rough estimate of relative time needed for each task. This is the schedule which will be presented in the proposal presentation on Thursday (figure 24). Any changes to this timeline will be noted as they occur. The presentation also requires an equipment and cost list, which is provided in list 1.

Figure 24: Proposed schedule

Date	Goals and progress
May-04	Decide overall purpose of the project
Jun-04 to Jul-04	Work on neural network framework
Aug-04	Redefine project goals and choose to use SNNS instead of new framework
Sep-04	Data processing functions designed and Chessbase 9.0 identified as database
21 Oct-04	Network generator program is created and data processing defined further
28 Oct-04	Order ChessBase 9.0 and evaluate training speeds
4 Nov-04	Argonne presentation and ChessBase 9.0 arrives, begin data extraction.
11 Nov-04	Extract Data, begin investigating possible network sizes and connectionisms
18 Nov-04	Extract Data, begin investigating radial basis function networks.
25 Nov-04	Extract Data, work on proposal
2 Dec-04	Extract Data, proposal presentation
9 - 16 Dec-04	Extract Data and begin to look at feed forward and radial basis comparison
23 Dec-04	Extract Data and work on rule logic and ANN integration module
30 Dec-04	Process Data (PGN to EPD), journal paper?
6 Jan-04	Process Data (EPD to FP), create and initialize all networks, journal paper?
13 Jan-04	Design a process for training and test on 4 or 5 machines, journal Paper?
20 Jan to 24 March-04	Train on maximum number of PCs, evaluate performance/make changes
31 Mar-04	Integrate remaining modules (final interface), test against human players
7 Apr-04	Continue testing system, evaluate rating if possible
14 – 28 Apr-04	Begin preparing for final presentations and expo + finish loose ends

List 1: Current equipment and costs

- **ChessBase 9.0 Database: $389.00**
- **DVDRs**
  - **Roughly 20 will be needed to backup data: $20.00**
- **160 GB external hard disk**
  - **For local data storage: $150.00**
  - **Personally purchased**
- **CPU Time**
  - **Off-hour access to laboratories will be required for training on as many PCs as possible**
  - **A full estimate of requirements will be made shortly**

Currently I do not foresee any modifications to this equipment list.

It seems that if any sort of spatial relationships are to be included in the training vectors, there are a couple approaches which may be taken. One of the more obvious approaches would be to consider the nearest threats to the piece involved in a specific move. Figure 25 demonstrates. The other approach would be to create a perimeter around the piece in question, which will cover some set number of squares. Any enemy pieces in these squares will be used as additional input into a network. Figure 26 demonstrates this. Unfortunately, as is clear from figure 26, this approach does not allow all possible threats to be taken into consideration, as the white knight in this example is not located within the "perimeter" even though it is a threat to the black pawn. In order to always be effective, the perimeter would have to include the entire board, which is essentially the same as considering the nearest threats.

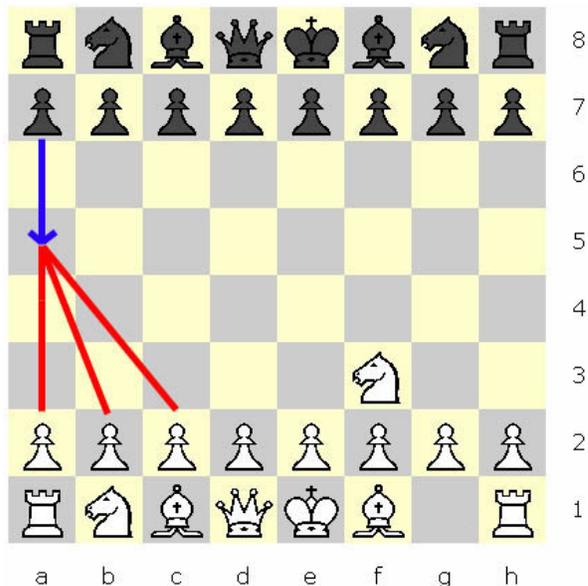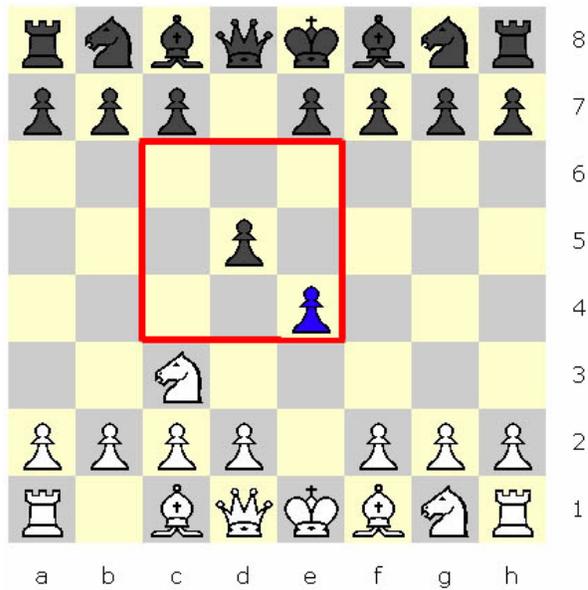Figure 25: Utilizing knowledge of nearest threats in training vectors

Figure 26: Utilizing a perimeter check in training vectors

It is also possible that the training vectors themselves would not end up being composed of floating point values at all, but instead binary representations would be used for each piece. This is somewhat like the approach described in the paper "Neural Network Learning In a Chess Endgame" located in the sources folder. The coding scheme used in this paper in shown in figure 27. As can be seen, this coding is essentially a form of one-hot binary.

Figure 27: Binary coding used in "Neural Network Learning In a Chess Endgame"
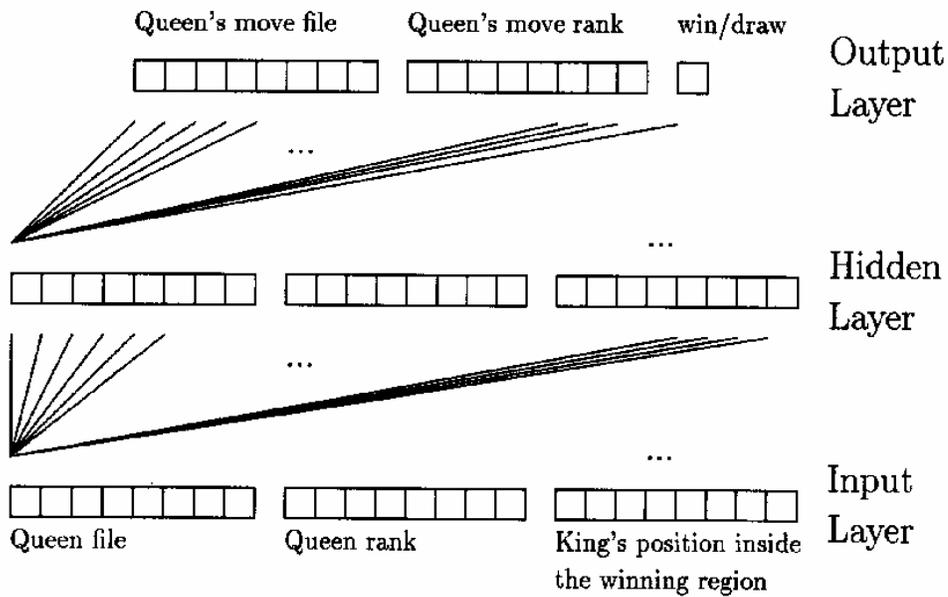
White Queen positions are coded as follows:
a → 10000000; b → 01000000; ...; h → 00000001;
1 → 10000000; 2 → 01000000; ...; 8 → 00000001.

These ideas will be presented with the proposal as possible alternatives to the main approach to be described (the geographical paradigm with floating point training vectors and feed-forward, hyperbolic tangent activation functions).

The rest of the day is spent creating the presentation, which can be seen in the journal files folder as "Proposal Presentation."

12-4-04

I have some concerns with the data processing. At this point, over 70 hours have been spent extracting data and it has only proceeded to about 50% of completion. There must be a better way to do this. I decide to try a somewhat new approach. I will extract every game in the database in which black wins or draws (1,978,263) in sets of 20,000 games. Roughly 100 files will be created in PGN format. I was able to locate an improved PGN to EPD converter application, called PGN2EPD.exe, downloaded from http://remi.coulom.free.fr/ . This application does not seem to be prone to the same crashing problem as PGNPosition.exe and may apparently process a list of 20,000 games without problem, but I still need to know what it does to games with errors (I will look at this shortly).

In any case, the idea is to write a batch file which will convert all games to EPD. Then, the EPD_FP.exe application I have created will be improved significantly in the following ways:

-Output decision based on move choice

-Text file load rather than copy/paste design

-Ability to work with file lists and move queues

Therefore, I will try to automate my floating point training vector creation program to handle batch processing of moves. My goal is to have the program create all datasets without supervision. I will need to produce a list of all moves for the program to parse when initializing searches. The additional functions which will be needed:

-Load file/convert to string

-determine "yes/no" output based on searched move

-move "parser"—obtain the move from a text file, then determine numeric locations for EPD string selection

-Method for determining "yes\no" decision ratio stored in the output. The program is designed to locate "yes" choices based on EPD string comparison. However, "no" choices to moves must also be placed in the output files. Four ways are considered:

1. Based on ratio selection (user input), wait until all "yes" patterns are found, then store "no" choices at the end of the file (make 2nd pass over the EPD file). Training will be set to use random pattern order.
2. Use a set ratio (50/50?) and randomly choose "no" patterns after each "yes" pattern is located. The file contains many more "no" for each "yes," so the ratio may be changed. This approach would "mix" the patterns while they are being created.
3. Save every pattern in every training set. The hard disk space needed would be enormous…so this may not be possible, even though it would probably provide the best data.
4. Only save "yes patterns" for each move. When training, *use all other datasets* to train the "no" patterns. Another application would be needed to switch the output value to "no" in these pattern sets as they would of course be stored with a "yes." Currently, I prefer this method as it would also use the least disk space and still allow all possible patterns to exist in each dataset. The final conversion of "yes" to "no" decisions in the training data would be done locally on a training machine.

The modifications to EPD_FP.exe are now started. The change is slightly more complex than anticipated as many functions used are inherent to the textbox control objects only, and new methods for handling text must be implemented in a string only version. The results are worthwhile however, as speed enjoys an enormous increase. I also add a progress meter so it is easy to evaluate how much time will be needed to complete a processing task. Currently, the application shown in figure 28 is capable of loading files, saving files, and deals entirely with strings. I am working now on the "yes\no" decision making, which involves determining the piece at the initial move location, then checking the next pattern's final location to see if the following two conditions are met:

1. The initial location must be empty
2. The final location must have the piece identified in the last string

If these are met, then the output is "yes." Otherwise, the output is "no." Programming for this functionality is near completion. The next stage is to add an automation "layer" which will parse a move file, and work with numerous EPD files stored in a single folder. I estimate the application will be ready within 5 hours of further work.

The source code thus far is seen in the journal files folder as "EPD_FP_041205.cs."

Figure 28: Initial screenshot of the updated EPD_FP application

EPD_EP

EPD: epdinput.edp

Name: training.pat

Find: a1-a2

Progress

Finding

Progress

Start   Save