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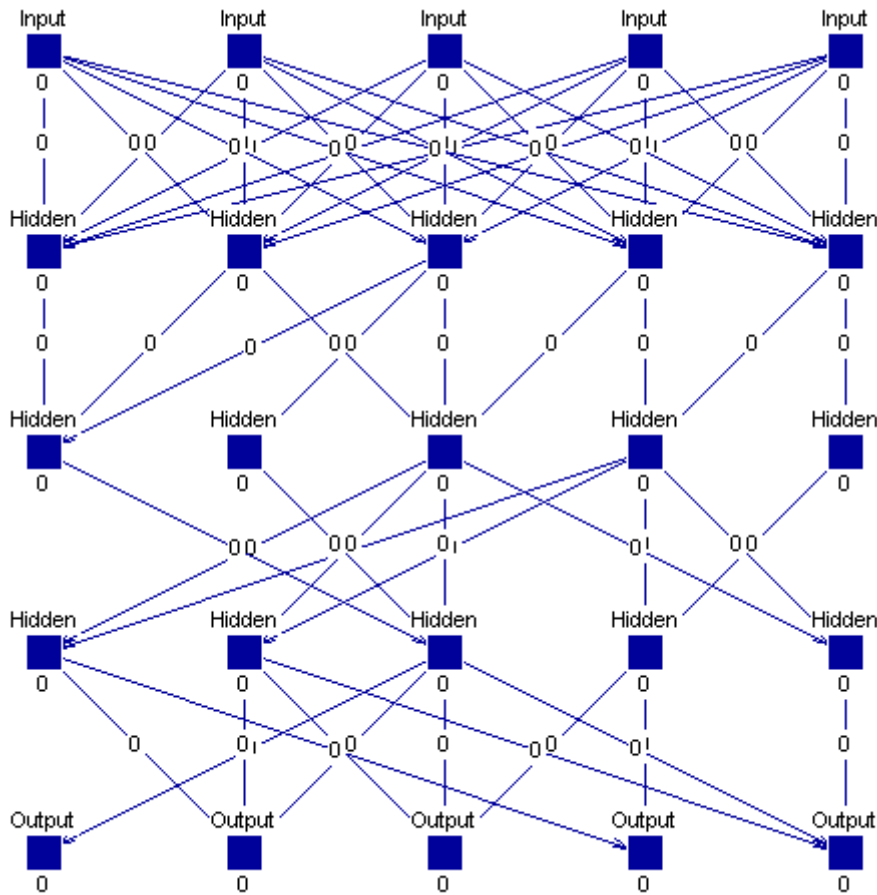
With the network generator now fully functional, it is time to start considering the data file processing. The dataset I will be using is capable of producing PGN (portable game notation) files, which are algebraic, standardized Chess game recordings. ChessBase 9.0 (NEED TO ORDER!!) should be able to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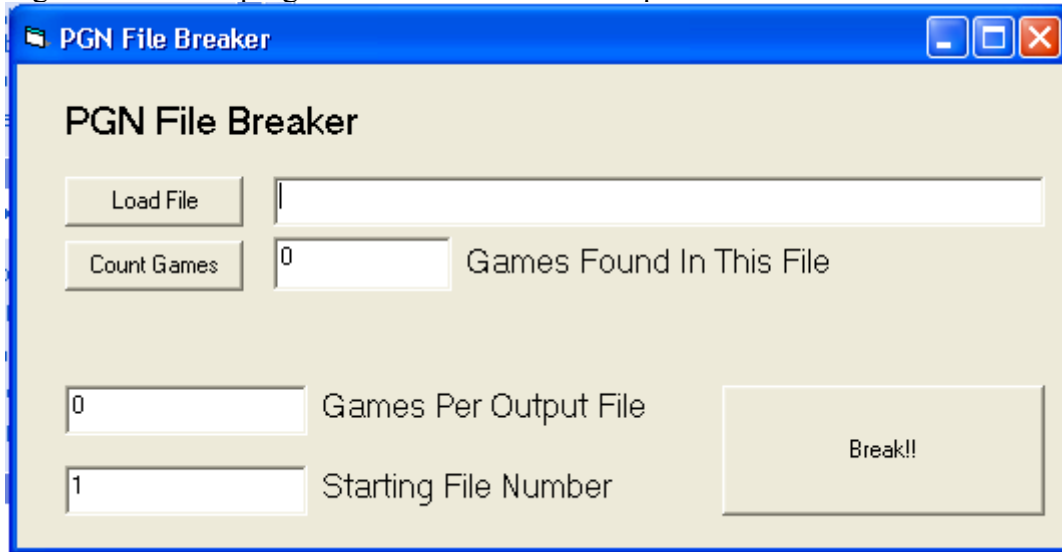
The PGN Breaker program is written in Visual Basic 6.0, and is simply a text parser. The code is shown in the Journal Files Folder as "BreakerCode." An example of the PGN format is shown in figure 5.

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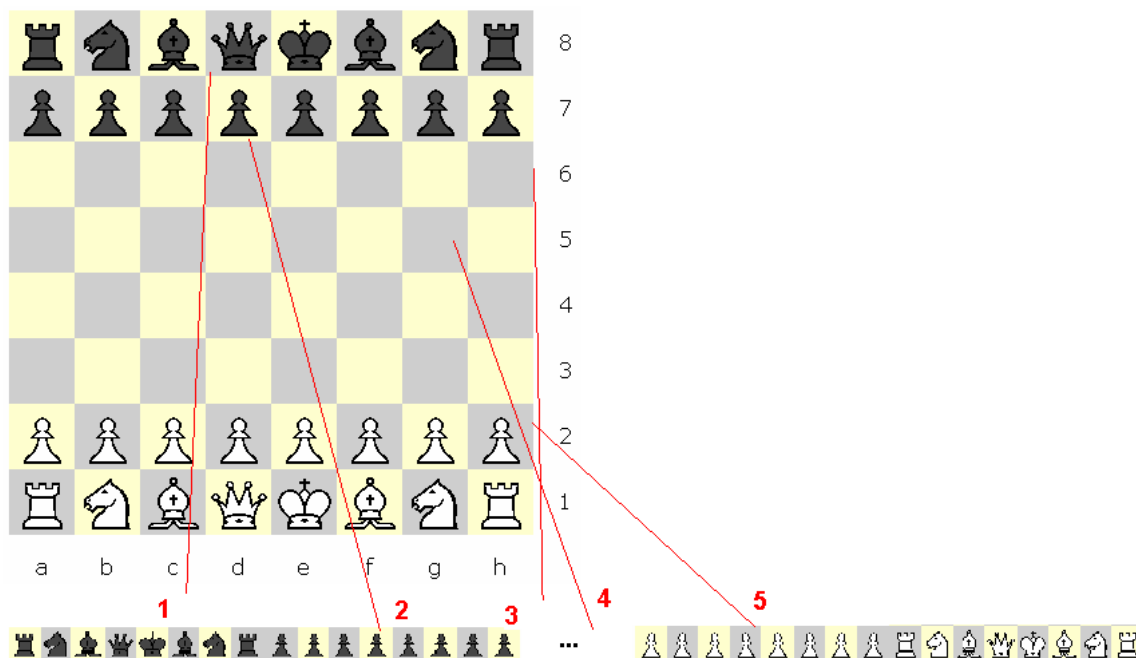
The inputs into the neural network will be in the same order as the positions are arranged for EPD format. Because the inputs to the network must be floating point values between

+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
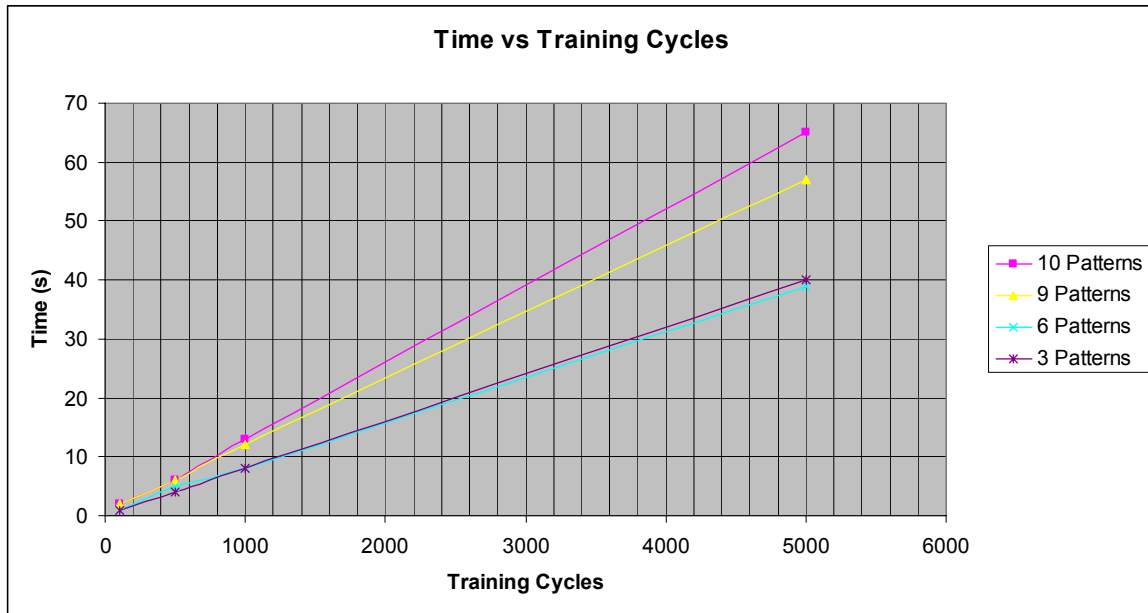
From figure 11, it seems that there is obviously a linear relationship between the number of training cycles and the time needed to complete them. There is no surprise here. But what about between the size of the training set and the time needed for a constant number of training cycles? Figure 12 shows 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
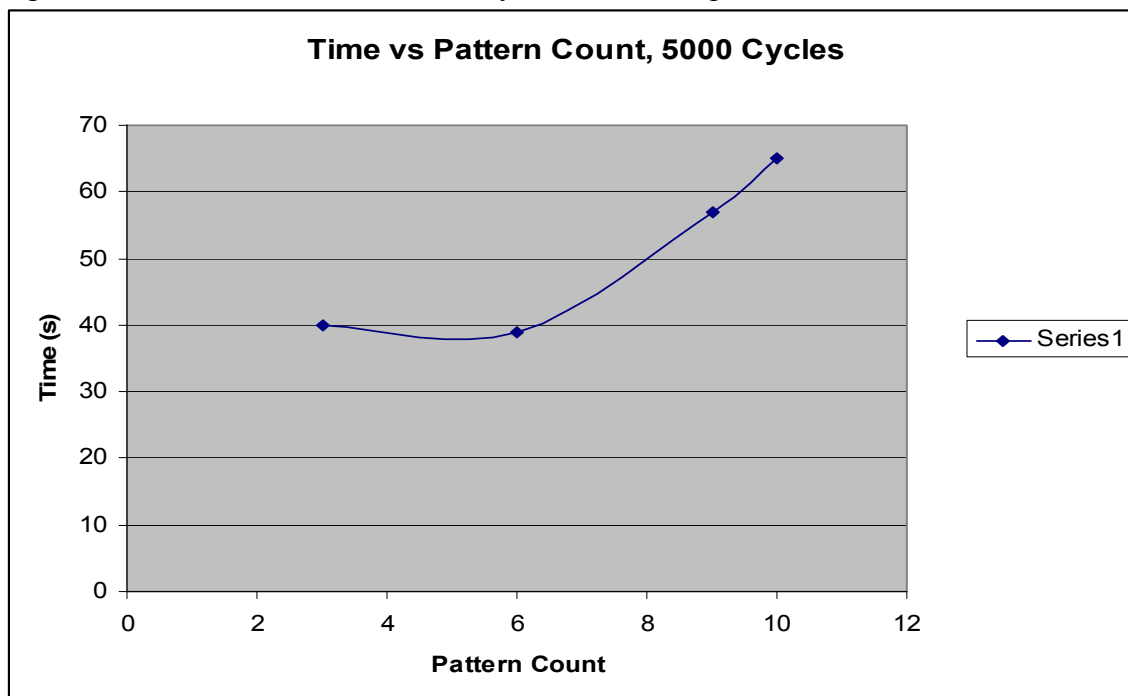
It seems that the pattern file size has a more non-linear impact on the training time required. Although only 10 patterns were tested as a max, it seems obvious from this plot that training time is going to be minimized by keeping a fairly small quantity of patterns in the training files. It may make 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
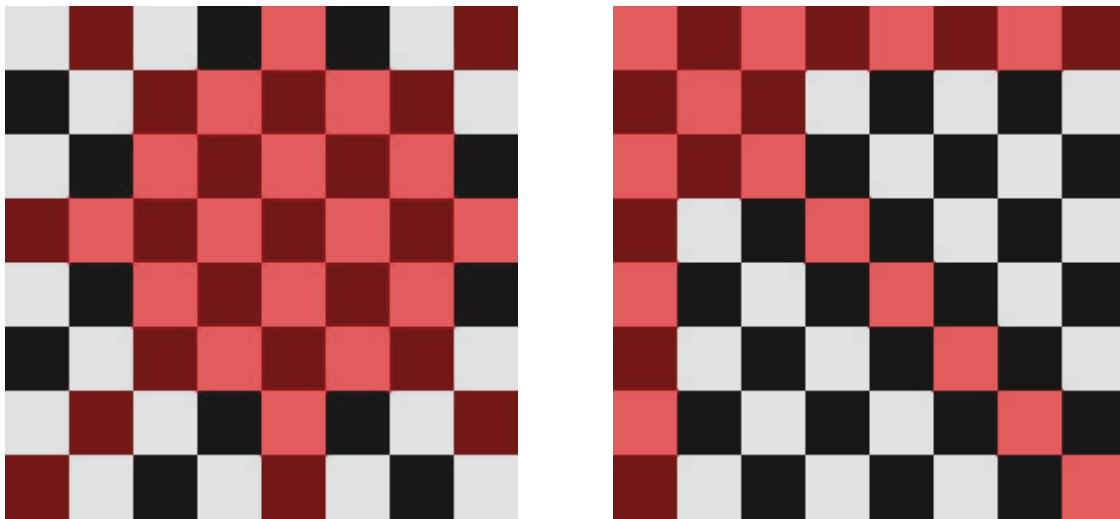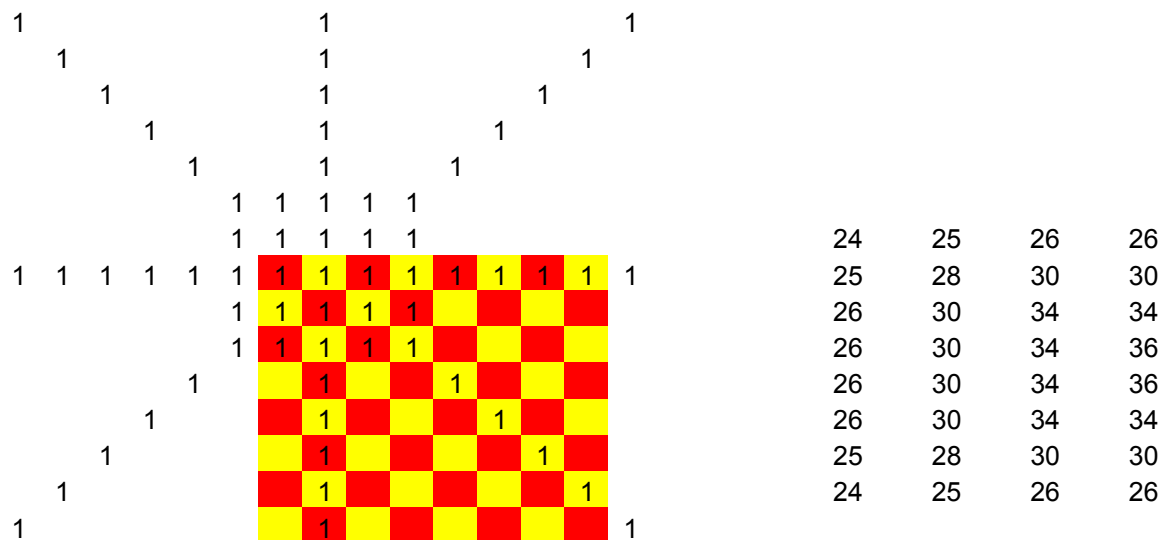
There are 1856 possible moves to be made at any given time. Thus, if I consider that 80,000,000 board positions exist in my training set, about 43100 positions would go into each category, taking roughly 2.5 hours to train. This is only for the "yes" decisions. An equal number of "no" cases would also 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

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
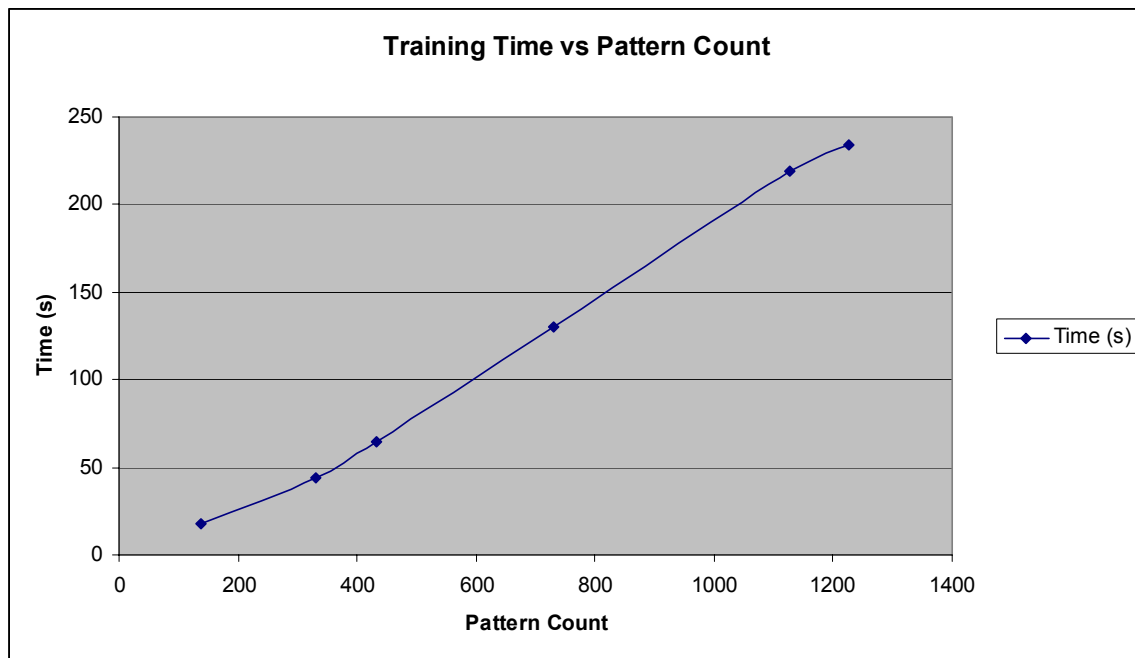
After completing the training, it appears that training time is actually a linear function of the pattern count, which was what the assumption wad to begin with. The discrepancy seen with lower pattern count is likely due to loading time, reaction time in starting and stopping the timer, etc. According 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

Training Time vs Pattern Count

$y = 0.186x$
$R^2 = 0.9819$
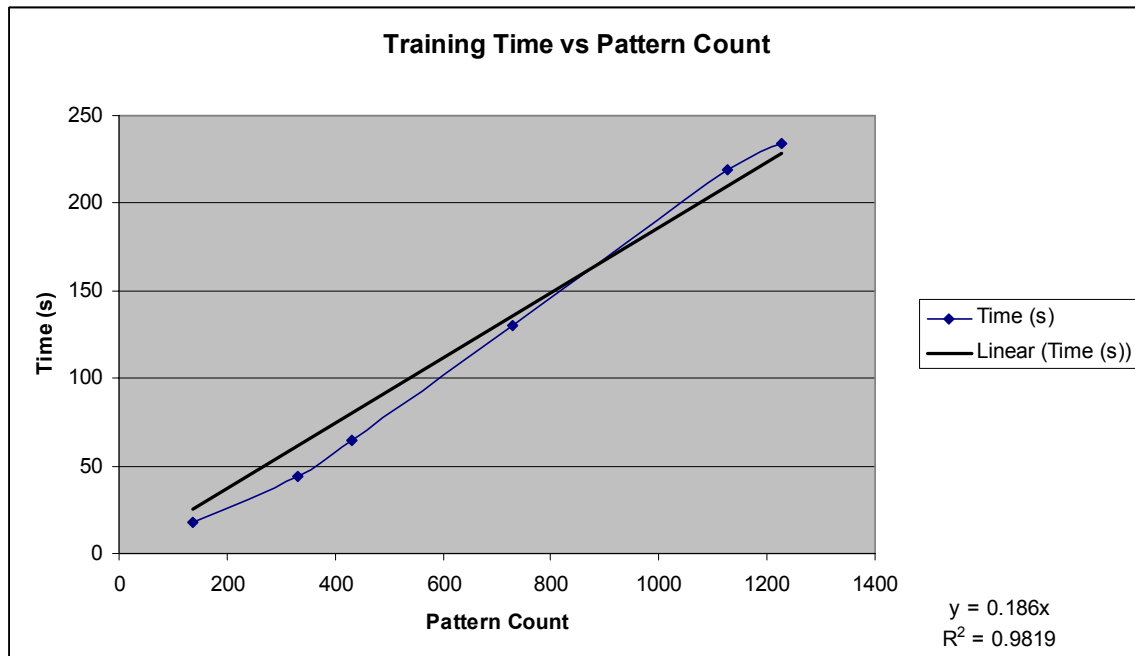
The R^2 value is close enough to 1 to have a large degree of confidence in the equation over the range of this data. Thus, each pattern will be considered to need .186 seconds to train.

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)
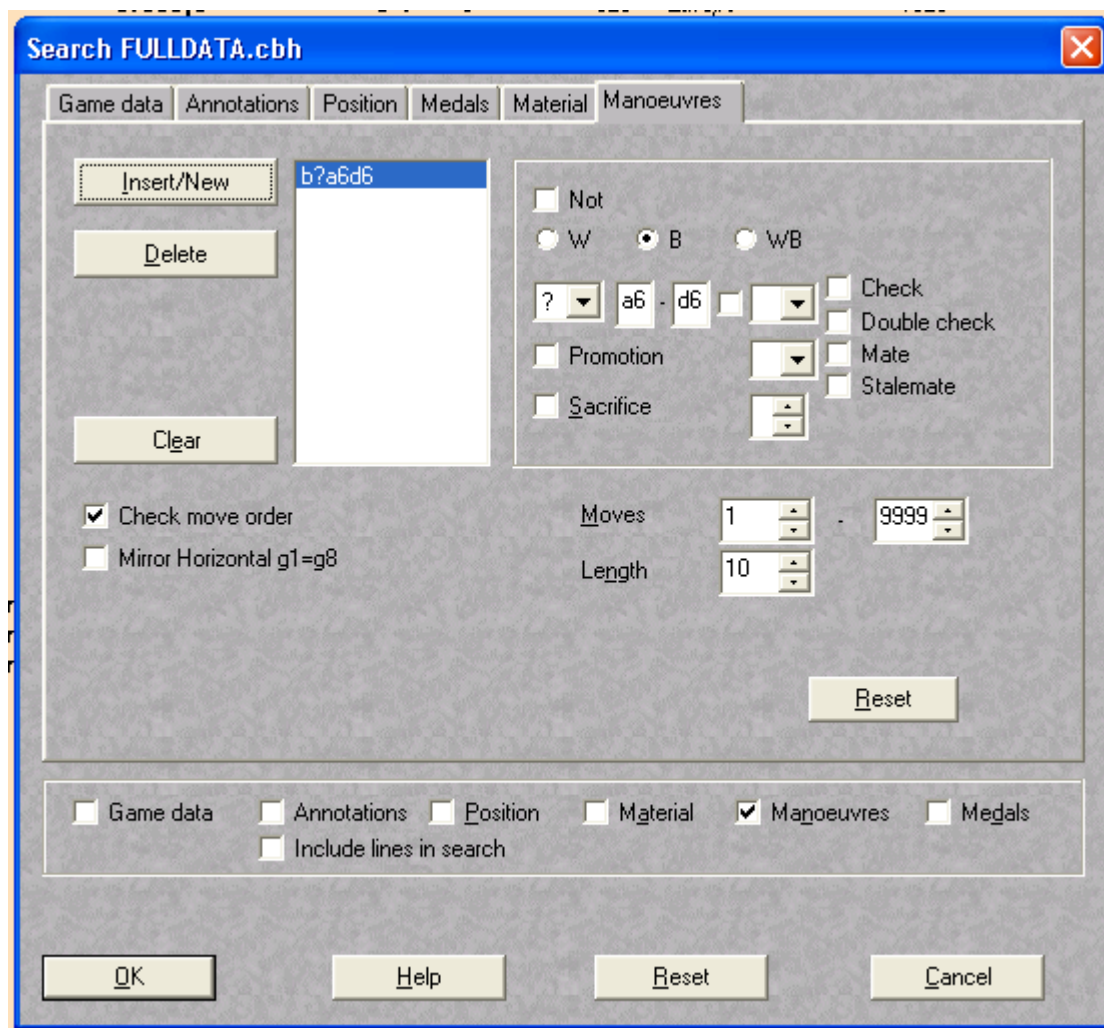
This same search will be performed for each and every move in chess, the total number which was calculated earlier to be 1856. After performing numerous saves, I find it takes roughly 1 minute for each move. About 30 hours (of manual searching) will be needed to collect all of the needed data. I install 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
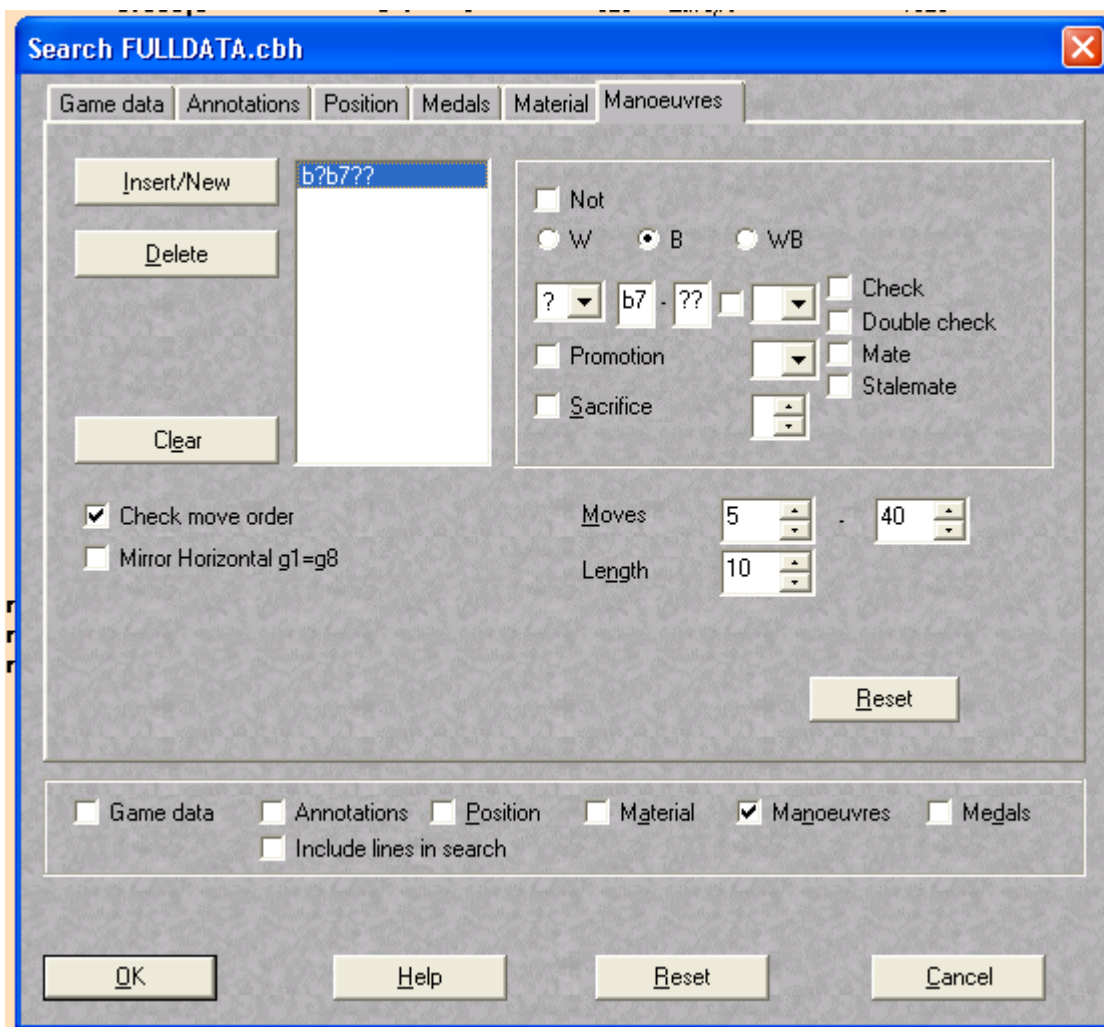
Today will be spent on data extraction, as it must be completed before any useful training can be performed. It may be possible to look at only one move, but evaluating this network as a standalone unit may be highly difficult, as we really need to evaluate performance over the entire game.

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
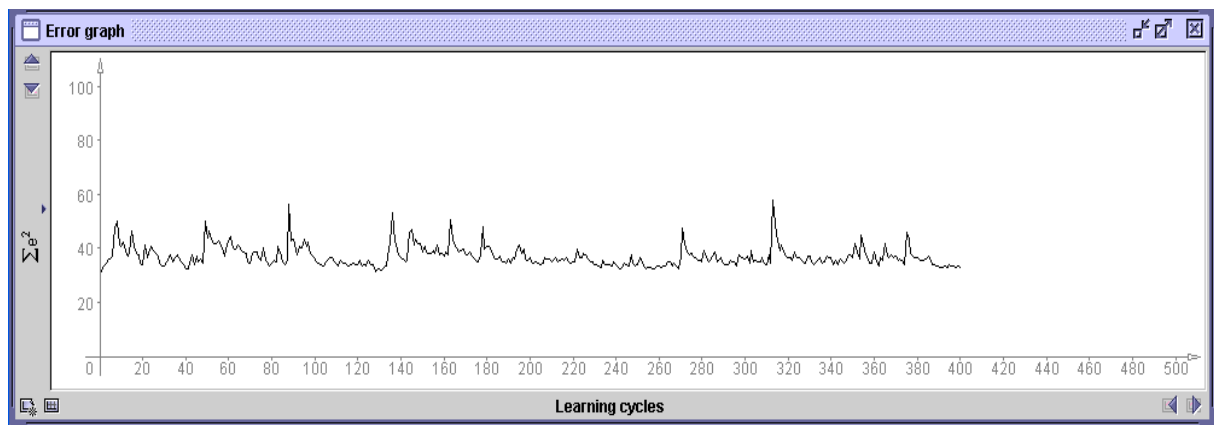
11-18-04

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).
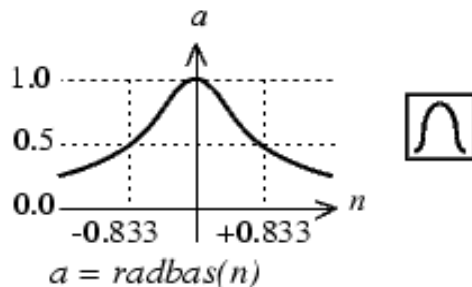
Radial Basis Function

-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).

$$a = radbas(\ \|\ \mathbf{w\text{-}p}\ \|\ b)$$
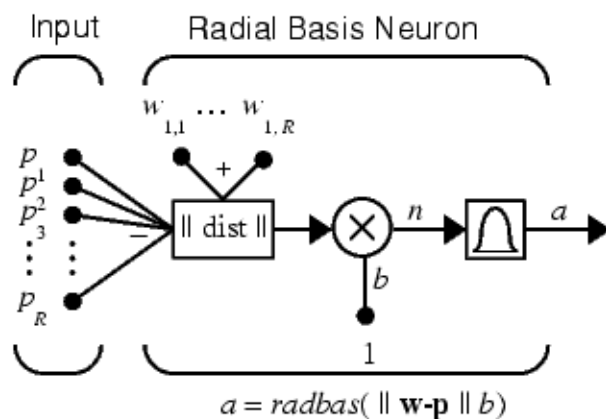
-The network will only have 1 hidden layer of nonlinear elements. An input layer and a linear output layer complete the network. Thus, a simple form of this network as represented in SNNS is depicted in figure 23.

Figure 23: SNNS representation of a simple radial basis function network
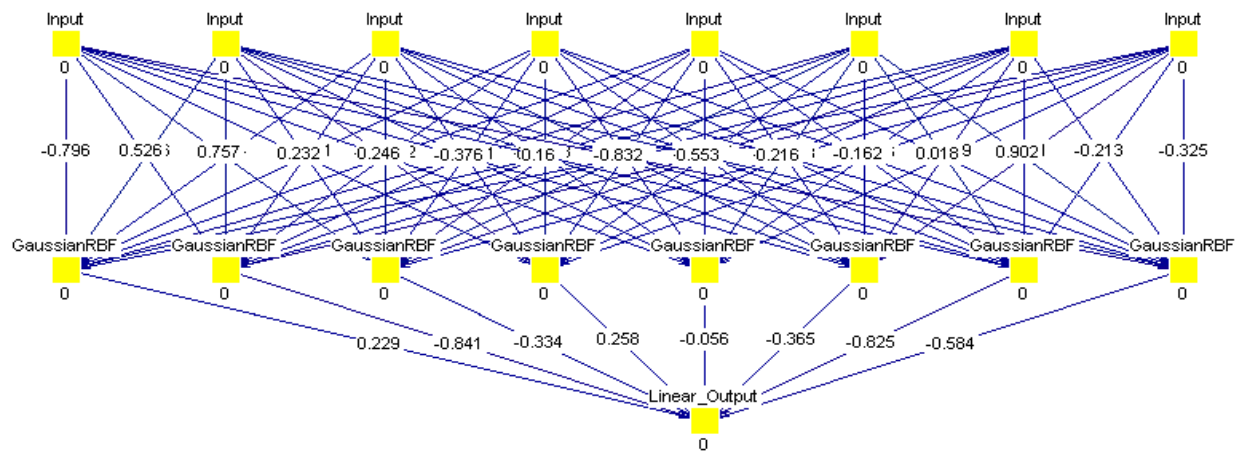
RBFNs are particularly promising for this project as they **are particularly good at pattern classification problems** if the problem can benefit from the fact that they make LOCAL APPROXIMATIONS or LOCAL CLASSIFICATIONS in each node (this is somewhat intuitive by the activation function's resemblance 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

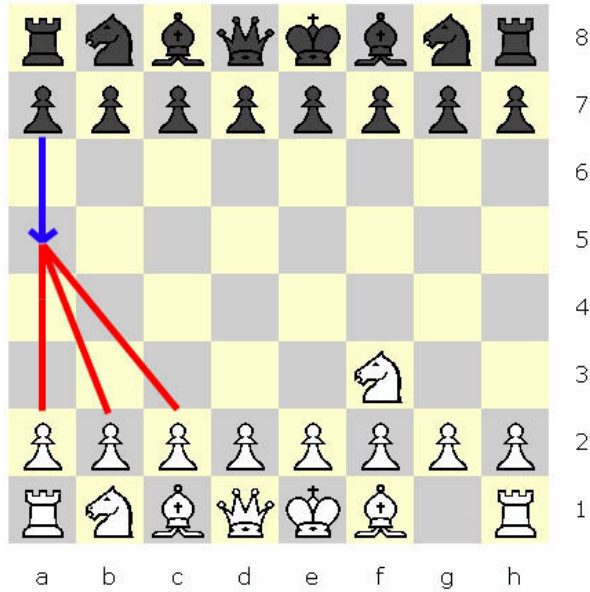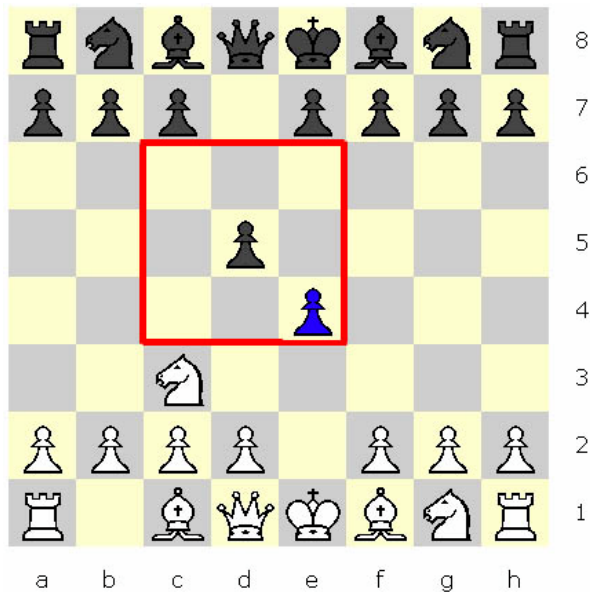Figure 26: Utilizing a perimeter check in training vectors

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"
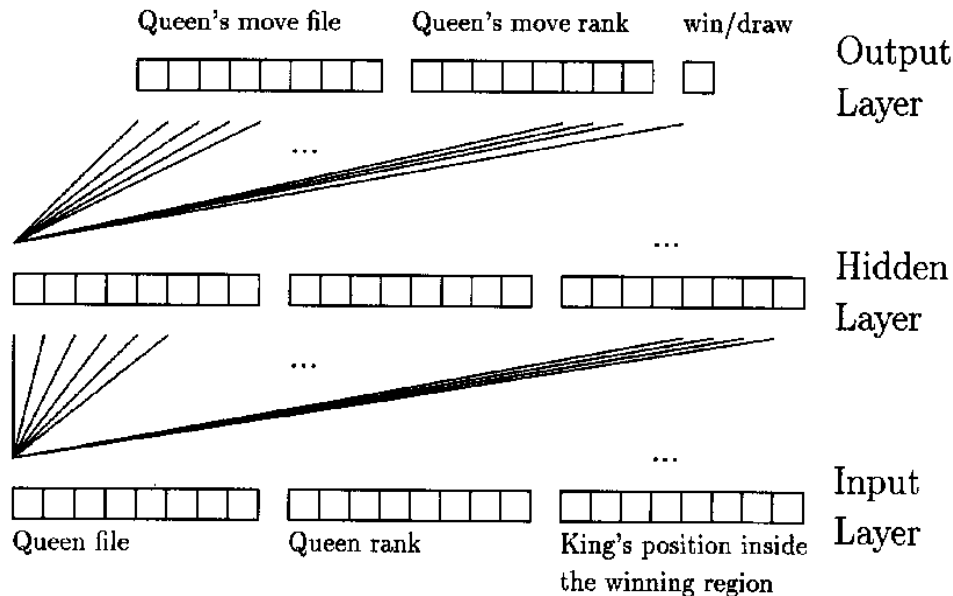
White Queen positions are coded as follows:
a → 10000000; b → 01000000; ...; h → 00000001;
1 → 10000000; 2 → 01000000; ...; 8 → 00000001.

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
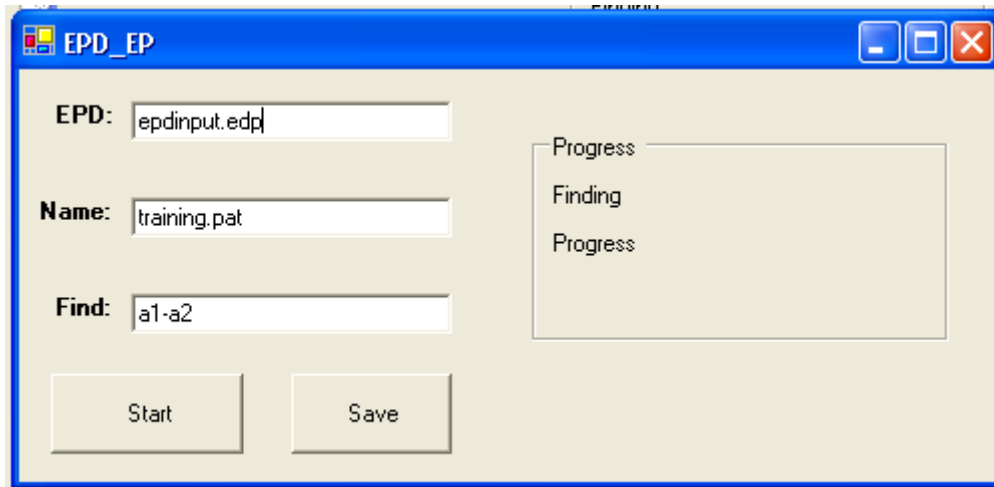
12-12-04

All games in the database are extracted and saved in PGN format. Sets of 10,000 games are taken, by game ID numerical order. These are backed up, and then three batch files are created to process the games into EPD strings. The batch files are located in the journal files folder, as PGN_EPDBatch1, PGN_EPDBatch2, and PGN_EPDBatch3. The entire process takes a couple of hours, but a full set of EPD strings has finally been obtained! Unfortunately, they are mixed up.

The EPD_FP program is now a console application, and all functionality described above has been implemented. The greatest problem is now execution speed. The source code up to this point is stored as EPD_FPV2_041212.cs in the journal files folder. I meet with Dr. Malinowski to find ways to make the code more efficient, and several improvements are implemented. First, the string objects (in EPD_FPV2_041212.cs) are changed to "stringbuilder" as an initial size can be specified. This prevents new strings objects from being created from existing strings whenever a new value is assigned. It seems this

improvement does help, saving about 5 to 7 seconds for each set of 10,000 EPD strings (a ".") is drawn every 10,000 processed EPD lines as a progress indicator). Each file contains about 400,000 strings, and there are 198 EPD files in all. Thus, total time savings from this improvement alone would be about 3.2 hours in all. It is also recommended that the "getposition" function (bottom of EPD_FPV2_041212.cs) be modified to work only with ASCII values, and not have the case statement currently implemented. This change is made, and the new function is shown in figure 29.

Figure 29: Improved function to get a numeric position from a row and file

```
private static int getposition(char file, char row)
        {
            try
            {
                int position=0;
                int file_value=Convert.ToInt32(file)-97;
                int row_value=Convert.ToInt32(row)-48;

                position=((8-row_value)*8)+file_value;
                //MessageBox.Show(position.ToString());
                if ((position>63)||(position<0))
                {
                    position=0;
                    //Console.Write("!");
                }
                return position;
            }

            catch
            {
                MessageBox.Show("Error in row, file");
                return 0;
            }
        }
```

Once again, a speed increase is realized, of a couple seconds per 10,000 EPD lines. In all, a few hours of processing time is saved by making this simple change. Although it is also suggested to use the close method on the streamwriter objects, further research show that "using" automatically implements the close method. It is therefore not required in this application. Another suggested improvement is to create arrays of streamwriter objects so that they may be kept open, which could eliminate the need to create them and also prevent the program from having to search for a file each time it is requested for writing. Clearly, major speed increases could be achieved by doing this, but here I decide not to carry out the implementation as processing time has now been cut down to about 2 days. It only has to be performed once, so there will be no recurrent time savings. Therefore, I consider the EPD to FP application complete. However, a few notes about the details of running it must be made:

1. The application is divided into 8 parts, which each process 25 EPD files (Except the last one, which only does 23). This improves performance as the output files get larger. This is also done because 24 GB of storage would be needed to save all files as once. I have tried the process working directly with the USB hard disk, but it is horrendously slow. It is better to simply move the data in stages.
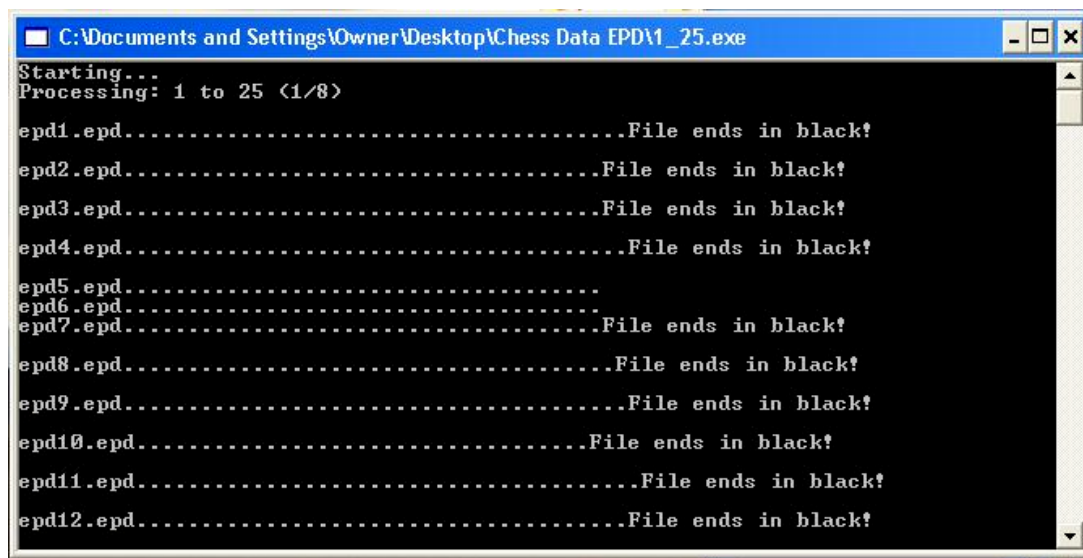
2. The .exe files must be in the same directory as the 198 EPD files to run.

3. All outputs are set to "-1". Another application will be developed shortly which will be run in the training stage in order to convert the "-1.0" outputs to "+1.0" for a given move. This application will also randomly choose patterns from all other pattern sets and save everything in a training file for presentation to SNNS. The whole purpose of this extra step is to save storage space. If all pattern sets were saved in there entirety (and non-compressed), nearly *48 terabytes* or storage would be needed! Only 24 gigabytes are needed with the current method!

The final source code for EPD_FP, version 2 is located in the journal files folder as EPD_FPV2_041213.cs. The 8 executable files are located in the programs folder, under EPD_FPV2. Also, PGN2EPD is located in the programs folder.

12-14-04

Now, the data processing is started again! This time however, it does not have to be supervised. All 8 EPD to FP executables are executed, an example of which is shown in figure 30.

Figure 30: EPD_FPV2 console window

This process is continued in the background, and other work can be completed at the same time. It will take a few days to finish this data conversion.

1-02-05

A training automation proposal was sent out around the last notebook entry. See CPUMemo in the journal files folder.

It has once again proven to take longer to work with the data than was expected, but the data has been fully processed at this time. The computer used for most of this data processing has been overheating fairly often, and turns itself off midway in the processing. It is required to restart in these cases to prevent the risk of a corrupted data file which may not load in SNNS when the time comes to start training.

In the next couple of days, all of the data will be uploaded to the GDANSK server. However, it is first required to remove any files which exist in the 8 data folders which do not represent legal moves. It seems that several mistakes existed in the raw data, as over 3500 moves exist in each folder! There should only be about 1800 in all. Therefore, it is required to write a program which will combine every initial position with every final position on the board and determine if the two constitute a legal move. The program will simply move the legal files into another folder, and the illegal moves will be deleted. This program's main legality check function will also serve as the legality check later on when the playing interface is created.

In order to make the legality check function, I create a class in C# which will be called "Position." A position has an x coordinate and a y coordinate, and 64 Position objects will be created in all. Figure 31 demonstrates the logic of the legality check.

Figure 31: Legality check logic
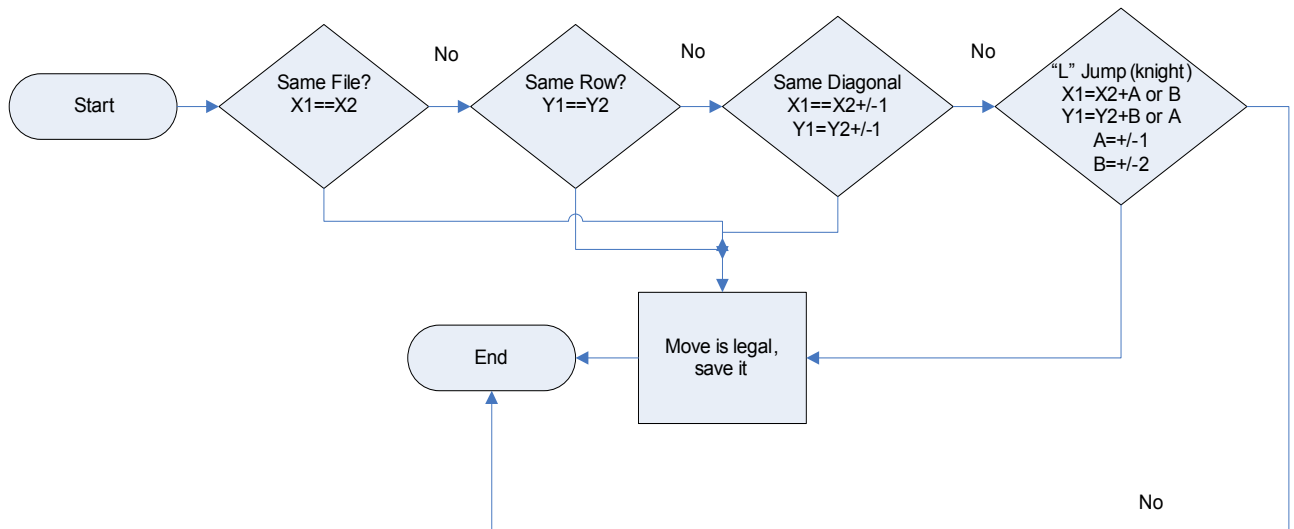
The program is created without incident, and is placed in each of the 8 directories containing .PAT files. After running, the valid files are uploaded to GDANSK. The source code for the program may be seen in the program files folder as LegalCheck.cs, and the Position class is included as Position.cs. It is likely that the position class will be used again shortly. The executable file is located in the Programs folder, as LegalCheck.exe.

1-10-05

There is now some concern regarding formation of the training sets themselves. Currently, all data has been sorted by move, but is stored in individual files. In order to train, samples from each file must be combined in some ration of yes/no decisions and the "yes" moves must have the output pattern changed to +1.0 from the current -1.0. A program will be needed to automatically do this, which will somehow fit into the proposed automated training process, which is described in figure 33.

The original idea for the learning automation was proposed in a memo sent on December 10, as seen under journal files "CPUMemo1." It was revised and the final version came out on December 13, 2004. It is available in the journal files folder as CPUMemo2. The text is shown as Figure 32, for ease of reference (it was omitted earlier).

Figure 32: CPUMemo2, requesting CPU time in the computer labs

ECE Department
Bradley University
December 10, 2004

To: Chris Mattus
        Dr. Aleksander Malinowski
        Dr. Brian Huggins

From: Jack Sigan

Subject: CPU Time and Laboratory Access Requirements

For my senior project, "Complex Decision Making With Neural Networks: Learning Chess," extensive computation time will be required. 1858 neural networks will have to be trained, and possibly multiple times. Initial predictions show that 100 3.06 GHz systems would need roughly 4 days to perform the proposed training in a maximum, worst case estimate (if 80,000,000 training patterns were trained on each network). It is hoped to drastically cut the time requirements, however it is clear at this point that many computers will be needed for the training stage.

The training procedure itself involves three components. The training environment runs in Java and is called SNNS (Stuttgart Neural Network Simulator), and is freeware. The second component is the training data which is currently being created. The third component is the automation layer, which will consist of a client application which will communicate with a central server. The client will download the training data and any configuration files needed to perform a training session. SNNS will produce a unique data file for each neural network trained, and after all training is complete the network files are to be harvested and integrated together in the final system.

During second semester, I am requesting provisions to perform this training on the computers belonging to the Department of Electrical and Computer Engineering at Bradley University. I am proposing that SNNS be available on as many laboratory machines as possible (whether locally installed or on a shared drive), preferably all Windows machines in Jobst 144, and Junior and Senior laboratories. The training data is to be downloaded as needed by each client machine from a single server. The initial space requirement per machine is estimated at roughly 5 GB of local hard disk space in a worst case scenario. The server will need roughly 50 GB of dedicated storage.

In order to perform training, it is proposed to take advantage of the time when the labs are closed, specifically at night between the hours of 11:00 pm and 7:00 am. The process will possibly require some human interaction to start (although ideally it will not), but will be fully automated after initializing. In short, the requested provisions are:

1. SNNS is to be installed or accessible on as many Windows machines as possible
2. Login privileges (depending on the extent of automation possible) and access 24 hours per day to all machines with SNNS
3. 5 GB hard disk space per machine
4. Installation of any automation/supporting programs required to perform training
5. Server space (already procured on "Gdansk")

The level of automation will determine the extent of access required during off hours. Ideally, a single task will be scheduled to run on the machines using Windows' task scheduler. The automation program, upon launch, will download a list of tasks from the server and will proceed to download the data files needed. Training will be planned to minimize download times, and training files will be stored locally once downloaded if they will be needed in the future. Once data is downloaded, the client will perform a simple output pattern remapping on the files based on the training assignment and then launch SNNS to start the training. "Key stuffing" will be required to allow the client to navigate the interface of SNNS automatically, and is currently the greatest technical challenge in getting the automation layer set up. If everything works as expected, the training will run completely unsupervised, and will not require a user to be logged in.

The full training plan will be developed by the early weeks of the spring semester and will be provided prior to starting training. Please contact me with any questions at 847-997-4402. It may be mutually beneficial to discuss this issue further in person within the coming weeks.

Figure 33: Proposed training automation process
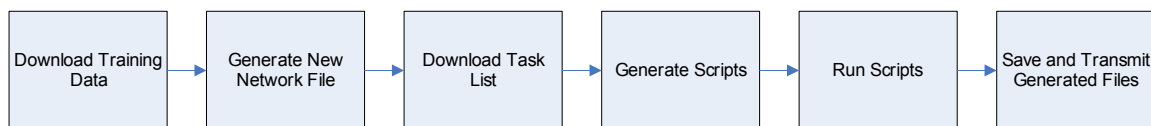
Before winter break, it was mentioned briefly that the training files must be composed of both yes and no decisions. Because the .PAT files are currently saved with only 'yes' decisions, it would be required to mix these with some 'no' patterns in order to prevent the network from reaching the incorrect, but simplest solution of always saying "yes" to any pattern presented to it. Earlier, it was stated that training could take place with all other files…unfortunately this does not seem to be a likely possibility. An assortment will have to be chosen to represent a sample of the total training data, as far too much exists to hope to train each network with everything. A method will have to be developed to solve this problem before the automation can be realized.

Also, it will be required to develop an application which will download all of the data off of the Gdank server and to a local client for training purposes.

It has been decided that training will utilize a scripting language for the majority of keyboard interaction steps required to start training, particularly in SNNS. A third-party scripting language is being utilized (found at http://home1.gte.net/res0mbu5/index.htm), which can be found in the Programs folder as Script. The language itself is described in the document SriptLanguageDescription found in the programs folder. Script.exe runs the script files, and must be executed via the command line. A batch file is created to launch a given script, and has the syntax:

script.exe AutoScript.scr

In order to launch SNNS, a script is created which will take the place of a person interacting with the keyboard to load a given training file and network, and start the training. Trial and error leads to a working version of the script, which is shown in figure 34. The script is executed by the batch file described above (a directory change is also required).

Figure 34: Script to load SNNS and start the training process

```
start,StartSNNS.bat              send,JavaNNS,\t
delay,20                         delay,1
send,Error message,\s            send,JavaNNS,\t
delay,450                        delay,1
send,JavaNNS,%w                  send,JavaNNS,\t
delay,50                         delay,1
send,JavaNNS,a                   send,JavaNNS,\t
delay,10                         delay,1
send,JavaNNS,^c                  send,JavaNNS,\t
delay,40                         delay,1
send,JavaNNS,\t                  send,JavaNNS,\t
delay,5                          delay,1
send,JavaNNS,\t                  send,JavaNNS,\t
delay,5                          delay,1
send,JavaNNS,\t                  send,JavaNNS,\t
delay,5                          delay,1
send,JavaNNS,\t                  send,JavaNNS,\t
delay,5                          delay,1
send, JavaNNS,\s                 send,JavaNNS,\<
delay,5                          delay,1
send,JavaNNS,\t                  send,JavaNNS,\t
delay,5                          delay,1
send,JavaNNS,\>                  send,JavaNNS,\t
delay,5                          delay,1
send,JavaNNS,\>                  send,JavaNNS,\t
delay,5                          delay,1
send,JavaNNS,\>                  send,JavaNNS,\t
delay,5                          delay,1
send,JavaNNS,\t                  send,JavaNNS,\t
delay,5                          delay,1
send,JavaNNS,\t                  send,JavaNNS,\t
delay,5                          delay,1
send,JavaNNS,\s                  send,JavaNNS,\t
delay,5                          delay,1
send,JavaNNS,\^                  send,JavaNNS,\s
delay,5                          delay,1
send,JavaNNS,\s                  send,JavaNNS,\t
delay,5                          delay,1
send,JavaNNS,\t                  send,JavaNNS,\s
delay,1                          delay,1
send,JavaNNS,\t                  send,JavaNNS,\t
delay,1                          delay,1
send,JavaNNS,\t                  send,JavaNNS,\t
delay,1                          delay,1
send,JavaNNS,\t                  send,JavaNNS,\s
delay,1                          delay,1800
send,JavaNNS,\t                  send,JavaNNS,%w
delay,1                          delay,50
```

```
send,JavaNNS,a                    delay,1
delay,1                           send,JavaNNS,d
send,JavaNNS,%f
```

The script is provided in the programs folder as AutoScrpt.scr. It is editable through notepad. In order to launch SNNS, the script calls another batch file, which is called StartSNNS (located in the program files folder). This batch file contains:

java -jar javaNNS.jar open network5x128_041028.net Test.pat Test4.pat

Of course, the file names network5x128, test.pat, test4.pat will be replaced in the final version with the real file names.

1-20-05

The focus today is to create the program which will create the training data sets from the .PAT files stored on Gdansk. The training files themselves will be 5000 patterns long. I have read that SNNS has problems loading extremely large pattern sets—hopefully 5000 is small enough. I decide that the training files will be created by taking 2 patterns from each of the 1791 pattern sets stored on Gdansk which do not match the "yes" move being trained. Therefore, 3582 patterns will have the "no" output trained, leaving 1418 patterns for the "yes" output. The problem is that all patterns must be chosen at random from within the individual .pat files. The overall process is described in figure 35.

Figure 35: Training file creation program
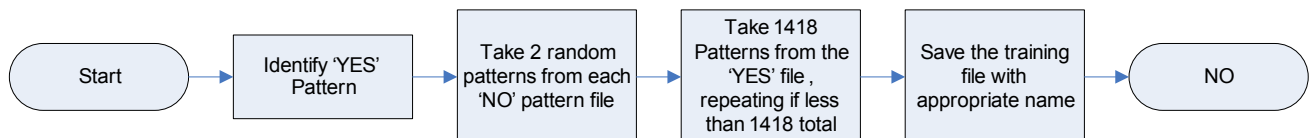
I write a program, called "merger" which should perform this function. The legality check function is used from the file moving program created last week to cycle through the list of legal move files. These files are opened one at a time with streamreader, and 2 random numbers are chosen which are less than the total number of patterns located within the given file.

Once the numbers are chosen, the program scrolls through lines until a counter reaches the lowest number of the set. The counter increments when the line "# Input pattern" is read. Once the first pattern is located, the pattern is read line by line and written to another test file using streamwriter. When completed, the .pat file is scrolled through line by line until the counter reaches the highest of the two random numbers. The counter does not reset after finding the first pattern. In the case of the "yes" pattern file, the number list contains 1418 values, which must be sorted prior to finding patterns. In the "yes" case, the program works in exactly the same fashion as it does with 2 patterns,
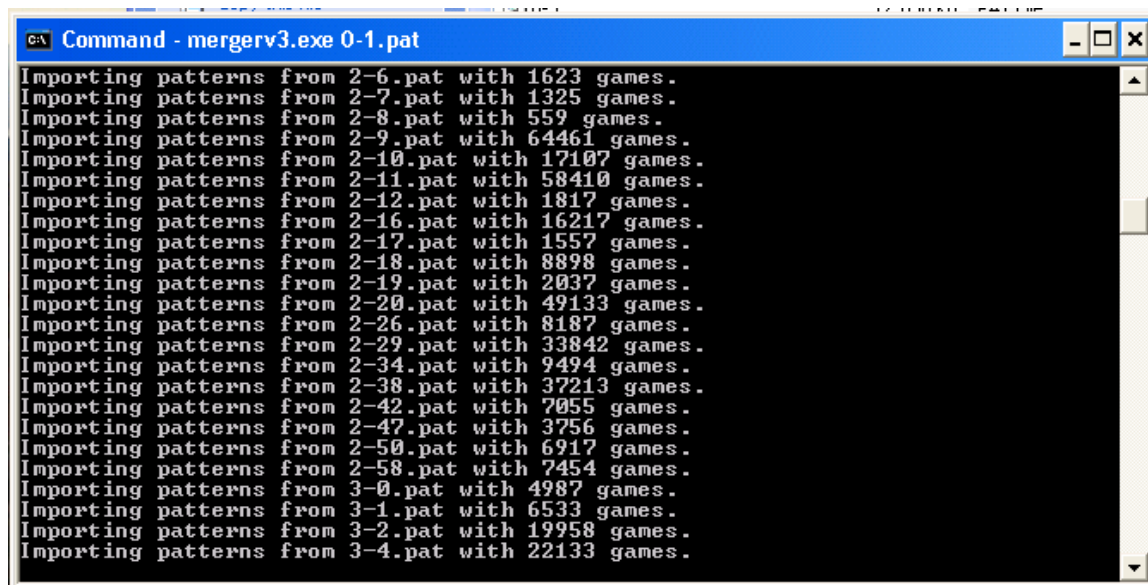
except 1418 are written to the training file, and the last line (output line) is changed to 1.0 instead of -1.0.

The source code for the final program is located in the journal files folder as "mergerv3.cs." The compiled program is located in the program folder as "mergerV3.exe." The program must be executed from the command line, and the command has the syntax:

Mergerv3.exe [yes-move]

Where yes-move must take a pattern file name, in the format [start-end.pat]. The resulting training file is stored in the same directory, and will have the name Train_start-end.pat. The program runs as shown in figure 36.

Figure 36: Mergerv3.exe running

Everything is now in place to actually start the training (at least manually).

I decide to train a network to see if learning can take place before continuing. I use the 5x128 network which was previously created, and create 2 training sets: one for training and one for verification.

From the training file creation, another concern comes up: the files are taking about 5 to 6 minutes to generate. I was hoping that all training data could possibly be created on a single machine and downloaded to each client, but this will be impossible given the time which would be required. Instead, it looks like the training files will have to be created on the client when training is to actually take place, after the raw data (.pat files) have been downloaded from Gdansk.

The two training sets now created are called: Train_0-1(Train).pat and Train_0-1(Verify).pat. The network file is renamed TestNet.net. In SNNS, all 3 files are loaded. All three files may be found in the journal files folder.

Before continuing, I need to create another simple program which will open the results files created by SNNS and perform a simple analysis procedure on this data. SNNS produces results files which have the format shown in figure 37.

Figure 37: Sample of results file format

```
SNNS result file V1.4-3D
generated at Thu Jan 20 15:23:00 2005

No. of patterns      : 1604
No. of input units  : 64
No. of output units : 1
startpattern         : 1
endpattern           : 1604
teaching output included
#1.1
1
1
#2.1
1
0.94571
#3.1
1
0.94571
#4.1
1
1
#5.1
1
-1
```
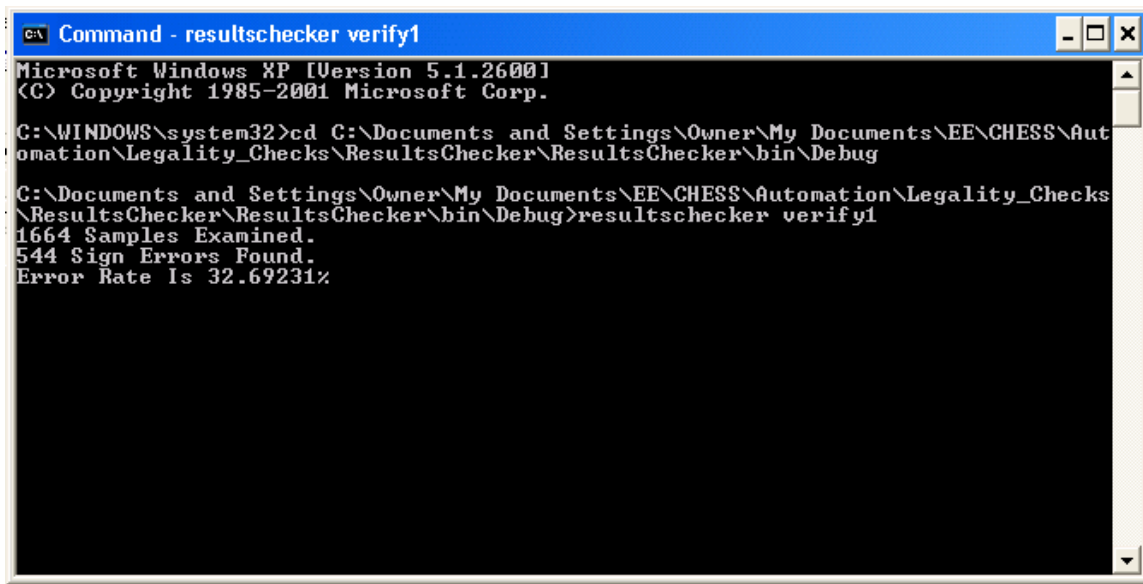
The program will simply locate a # symbol, and then read the next 2 lines. The first line is the desired output while the last line is the actual output. The signs will be compared in the initial version. Each time the signs do not match will be considered an error, and the total number of errors will be returned along with the error percentage. Although very crude, this method will allow a basic determination as to whether or not training is actually occurring. The program is called ResultsChecker, and the source code may be found in the journal files folder as ResultsCheck.cs. The executable is located in the programs folder. The program is capable of running with a command line argument or by itself, in which case it will ask for the name of the results file to verify.

Figure 38 shows the program execution with a command line argument.

In the future, it will likely be required to improve this application and add functionality to check the amount of divergence from the ideal value, and possibly other still unknown characteristics. More will be discussed as the need arises for improved results analysis.

The file loaded in figure 38 is called verify1. It does not have any meaning so it is not being included in the journal files folder.

Figure 38: Results Check application running with a command line argument

```
Command - resultschecker verify1                          _ □ ✕
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS\system32>cd C:\Documents and Settings\Owner\My Documents\EE\CHESS\Aut
omation\Legality_Checks\ResultsChecker\ResultsChecker\bin\Debug

C:\Documents and Settings\Owner\My Documents\EE\CHESS\Automation\Legality_Checks
\ResultsChecker\ResultsChecker\bin\Debug>resultschecker verify1
1664 Samples Examined.
544 Sign Errors Found.
Error Rate Is 32.69231%
```

Now that a method exits to actually analyze the training results and progress, the network **TestNet.net is initialized in SNNS, and then results files are created for both pattern sets** previously mentioned.

The results files indicate that the "train" data produces 52.2% error, and the verify set is producing 56.82% error. This is without any training, so the result is expected to be about 50%, since yes or no decisions would be evenly distributed in an untrained network. Now, the network is trained with the "Train" file, for 100 cycles, using Resilient Propagation as the training algorithm. It has been observed that this training algorithm produces the most rapid change in the error plot, after trying all algorithms. The difference between Resilient Propagation and regular BackProp must still be investigated. Default settings are used, but the "shuffle" option is chosen for the training pattern order. After the training is completed, the results file is saved (with output patterns) and the error is determined to be 8.5%. Clearly, learning has taken place. How well will the verification set do? I load this is SNNS, and save the results file. The error is determined to be 13.2%. Although this is higher [possibly] than desired, it does at the very least serve as evidence that the learning is indeed taking place. All four results files (pre and post training) are located in the journal files folder, as Train_0-1(Train)_Results, Train_0-1(Verify)_Results, Train_0-1(Train)_Results2, Train_0-1(Verify)_Results2.

Further questions certainly remain regarding the training procedure. These will be addressed in the near future. The learning automation program must also be completed so that training may begin on several computers at once.

1-25-05

Prior to considering training details any further, I will be working on the application which will be used to download the raw data files off of Gdansk. The raw data files are the .pat files used by the merger application to generate the final training files.

Gdansk has 8 folders which each contain enough data to generate a training file. In other words, every one of the 8 folders contains different versions of the same files. I decide that the training files should be as "random" as possible, so the downloader application will construct a full set of files by downloading from the 8 directories randomly. Once all 1792 files are downloaded, the application will exit. Essentially, this program is exactly the same as the program used in the file moving application made earlier. The only difference is that the line which moved the file is now changed to create a WebClient object and then call the DownloadFile() method. The source code is in the journal files folder as Downloader_050125.cs. The executable file is located in the programs folder (as DataDownloader.exe).

The question which now comes up is whether or not the training data is being chosen in a "proper" manner. The concern is that the "no" positions may not provide an effective sample to use in training. The problem arises for the simple fact that all 'yes' decisions have a common trait: There is a piece present at the start position. The 'no' positions have a majority of this same start position as empty. I am concerned that the networks may make the incorrect association that a piece being present automatically would translate to a "yes" move. It may be required to also train the networks with a higher ratio of "filled" start position 'no' patterns in order to achieve a quality generalization. After talking to Dr. Malinowski, we have agreed this is indeed the safest solution. I will now be modifying the MergerV3 application to create a pattern mixture. ~~The goal will be a 50/50 mix of filled and unfilled positions, although this is only a guess~~. It may prove necessary to adjust this number later on once training actally begins.

After further consideration, I decide to create the training files by looking at the initial position in the file name. If this matches the "yes" move, 100 patterns will be copied from the file to a training file. The .pat file which contains the "yes" move will have 1000 patterns copied. The total number of patterns in the training file will be around 5000, but will vary based on the number of files which share the initial position. This decision was made mainly for the speed advantage, and for the fact that most occupied initial positions show up in the files which share the same start position. When looking at files with a mismatched start position, it seems the desired square is occupied very infrequently, perhaps as low as 2% of all patterns, which is based on examination of a few .pat files. I feel this infrequent occurrence justifies omitting the files from the training set creation. By only opening the pattern files with the same initial position and copying a relatively large number of patterns from them, the execution time of the program is dramatically reduced. While it is true that this is not a totally ideal training set, I do feel enough files are going to be involved in the training set creation to produce a fairly representative training set.

1-27-05

The new version of the Merger program is in the programs folder as Mergerv3(final).exe. The source code may be seen in the journal files folder as "Mergerv3(final).cs". I create

several datasets and open them with SNNS to ensure that the files being created do not have errors in them. All tests are completed successfully.

Now, another speed improvement becomes possible for the training program. Because the training files are based only on the .pat files which share the same starting position as the "yes" move, it is possible to dramatically reduce the amount of data required for training. In fact, only the pattern files with the particular starting position in their name will be required. I will now work on a schedule for training. I will assume 14 PCs will be available in Jobst room 248. In addition to this, I will provide 2, Dan Leach will provide 2, and Dr. Malinowski will provide 2 as well. This gives 20 PCs in all which can be used in training. To start, I will set the schedule up so that each PC will create all training files of a given starting position. Thus, 64 sets of training exist to be divided out between PCs.

Currently, the list of computers names available are:

Job248a
Job248b
Job248c
Job248d
Job248e
Job248f
Job248g
Job248h
Job248i
Job248j
Job248k
Job248l
Job248m
Job248n
ECS732g
?1
?2
?3
?4
?5

There are 5 PCS which have unknown names at this time. Figure 39 shows the training schedule. Modifications will be made if the number of available PCs changes. Training will be done in rounds. Each round will train each network with one set of training data. In all likelihood, it will be required to run several rounds of training to achieve quality playing performance. Each round of training will have an essentially random set of training data (as described with the merger program).

Figure 39: Training schedule (showing starting position in each set)

job248a	job248b	job248c	job248d	job248e	job248f	job248g
0	3	6	9	12	15	18
1	4	7	10	13	16	19

2	5	8	11	14	17	20

job248h	job248i	job248j	job248k	job248l	job248m	job248n
21	24	27	30	33	36	39
22	25	28	31	34	37	40
23	26	29	32	35	38	41

ECS732g	?1	?2	?3	?4	?5
42	49	52	55	58	61
43	50	53	56	59	62
44	51	54	57	60	63
45					
46					
47					
48					

I decide to make task lists for each PC, which will have a list of the training tasks. The tasks will not be date-dependent, but will instead be executed in sequence as quickly as possible. After one training task is complete, the next task will start after it is verified that enough time is available. When a task is complete, the task will be placed in a local data file which will hold all completed tasks. This file will be emailed to me each day so I can track the training progress. I will also have the completed network files mailed to me. The newest version of the automation process is shown in figure 40.

I have discussed a potential problem with Nick Schmidt this morning. Yesterday I realized that all PCs will need the .NET framework installed in order to run this automation process. The PCs in the lab do not currently have .NET installed, but it appears that it is possible to install in fairly quickly with a "bootstrap" version. It may also be possible to install it on an image and re-image the PCs in the lab in the morning or on a weekend. Nick does not feel there will be any problem with this. I will be talking to Mr. Mattus, Nick Schmidt and Dr. Malinowski later today to make arrangements for setting up the training programs.

Figure 40: Automation process flowchart
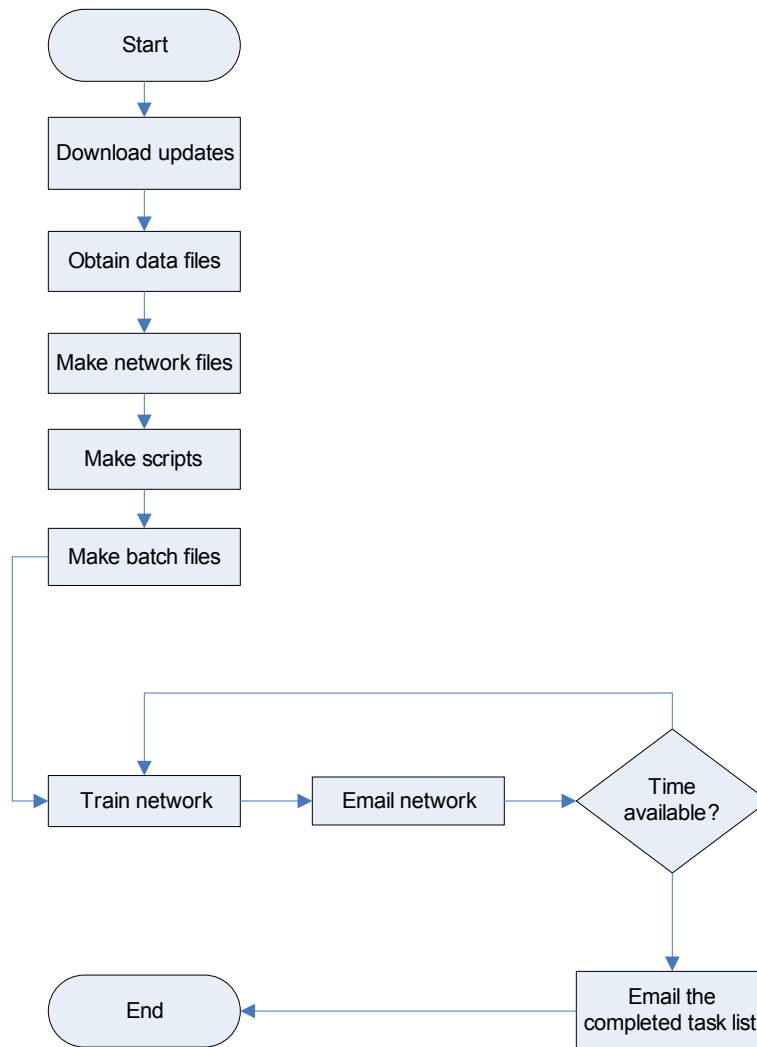
Talking with Mr. Mattus proved somewhat disastrous as he refuses to install the .NET framework. Although he has agreed to allow the installation of the various training programs, most of the software is useless without .NET. Before training can begin, I must consider my options for making the modifications to the training process.

1. Programs can be rewritten in C++ so they will run on the lab PCs. Data can be processed on the server to produce training files so raw data is not manipulated on the clients. The training program will only launch SNNS, send files, and generate training scripts.

2. The C#.NET approach can still be used, but the lab PCs will not be used. Training will be performed by volunteers on their own PCs.

3. Some combination of the two can be used.

Currently, I am leaning strongly towards option 2. This would allow far greater flexibility in training software, and would allow training to be started more rapidly. Of course, it would be best to use the lab PCs and the volunteer's PCs. However, the lab PCs will require that software be rewritten—there does not seem to be a way around this fact.

2-1-05

At this point, I am working to complete the software in C#. Several people have offered to run this software on their personal PCs, which will have several benefits. Mainly, screensavers and power save features can be disabled. After running the SNNS automation on the lab PCs, I find it does not reliably start and complete the tasks, although I am not sure why. All programs start, but the script does not seem to activate the training process. Most of the "volunteer PCs" are newer and "faster" than the lab PCs as well, which may lead to a slight improvement in training speed.

If time permits and training would benefit a great deal, I will probably modify the software by rewriting in C++ once training has started.

The updater application is now complete, which operates according to the flowchart seen in figure 41. This application is the first to run in the automation process, and will simply write over any existing applications which are to be updated (a list is downloaded). All updated applications will be stored on the Gdansk server. The update application is called Updater.exe and is found in the programs folder, and the source code is named Updater.cs and is found in the journal files folder.

Figure 41: Update process flow chart
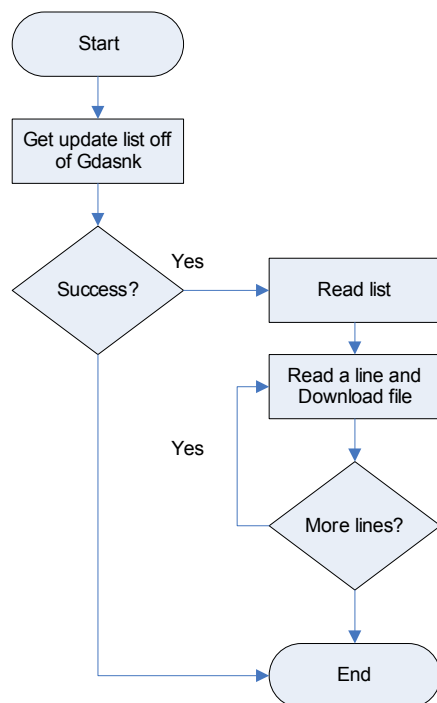
Now, the next requirement is to create the process which will determine the data files to download in order to create the training files. The training scripts will also need to be created.

Before continuing with this, I look into the various methods which could be used to transfer the completed network files back to me. I rule out e-mail for the reason that an SMTP server will have to be specified on a system not running Windows 2000 or XP professional. I decide to use FTP based on the extremely simple process—I will probably use a script and use command line FTP to send the files.

The question which remains is what FTP server to use. Typsoft.exe is a good FTP server which I could run on my PC, and provide write access to one folder only for all clients. Another option is to use Gdansk, and provide write access to only one folder as well. There may be some security issues with this, so I will have to talk to Dr. Malinowski. The other solution is to use a third party hosting service, and pay about $3.00 per month. This may be the best solution, as it eliminates the security risk to my personal computer and Gdansk, but does not lose any functionality. I am also assured it will be online at all times.

I will be looking into the best solution to this problem some more, after getting advice from Dr. Malinowski and other students. My reservation with using Typsoft.exe is twofold: I do not want to grant access to my personal computer and open it up for security issues, and I also do not want networking services calling me about running a server on the campus network.

I need to solve this problem first, because I will need to somehow assign computer IDs which will determine the training schedule assigned to each client. My idea is that each client, when they start to run, will generate a random number and send it to the server. This number will serve as their unique ID. Each client will download training schedules with file names matching this number. Thus, each client will have a "configuration file." The other option is to make unique copies of installation CDs which will have a custom configuration file already built in with a training schedule.

For the time being, I create two files. One of them is called "configuration.dat." This file will contain a list of networks which will be trained. The other file is called "finished.dat," which will contain the names of networks already trained. As training proceeds, the automation software will check to see if the network name exists in the finished list. If it does not, then a new network will be created, trained, and sent via ftp back to me. After training is complete, the network name is added to the finished list.

I now modify the network generator program so it will be easily automated. It may eventually be converted to a command line application, but for now it will stay as the GUI application. The changes are seen in figure 42. The main change is that all parameters are now specified, and the tab order of the bottom three buttons was changed to 0,1,2. I also added the quit button, so exiting the program is easily automated. The new source code is in the journal files folder as netgen2.cs, and the application is found in the programs folder as NetGen2.exe.

Figure 42: Modifications to the network generator program

2-3-05

The overall automation "shell" will be created today. With any luck, it will be working by the end of the day. I am pretty sure all components required are now ready, so the automation process described in Figure 40 should be attainable.

The shell will first check for the next network to generate. The name is found by looking in the "finished.dat" file for the next entry in the "configuration.dat" file. If it is not found, then a network is created and the training process is started.

After a network is found, the shell will create all of the training scripts and batch files. This is simply done by using streamwriter to overwrite the existing files, which will of course contain paths and file references to the network file mention above. The scripts and batch files which need to be created at each training round include:

StartSNNS.bat:
cd C:\Documents and Settings\Owner\My Documents\EE\SNNS
java -jar javaNNS.jar open network5x128_041028.net Test.pat Test4.pat

FTP.scr:
start,ftpcmd.exe
delay,20
send,*ftpcmd*,open 65.13.58.239\n
send,*ftpcmd*,jsigan\n
send,*ftpcmd*,*******\n

sned,*ftpcmd*,put xxxxx

RenameNet.bat:
Rename [test1.net] [test2.net]

DataDownload.bat:
DataDownloader.exe [network name]

merge.bat:
mergerv3.exe [network name]

Basically, each file is created to match the name of the network to be trained.

The code for Autoshell is found in the journal files folder as Autoshell.cs. The application is found in the programs folder. All that remains is to link everything together one last time and create a program to update the finished files list after a successful network transmission back to the server.

2-10-05

I now am working on putting a distribution together, which will be easy to install for all users. I will also include the java run-time environment in case a user does not have it installed. I have decided to make the learning "program" available for download, so I will hopefully find several users willing to install it. I have a couple of remaining concerns:

1. How will the user get the list of files to train?
2. How to safeguard my password to the upload server?
3. How will the process start again once it is complete?

Luckily, my Updater application will allow me to play with #1 after the release. The first users to agree to installation will get personal training lists. If enough users agree to install it, I will release a "patch" which will generate the training lists automatically.

I modify the paths within all of the .bat files to work off of the hard disk. However, when I do this, the SNNS-starting application (and the automation script) seem to be out of sync. SNNS starts, but the script does not perform correctly. Instead of going through the prescribed sequence of keystrokes, the program goes out of control, and unpredictable keystrokes seem to be returned! Obviously, training does not start correctly. It takes me some time to find the problem, but I realize that within the AutoShell.exe application, I am writing the startSNNS.bat file with a path specified as "C:/autolearnrelease/" which is resulting in an error. I change it to "C:\autolearnrelease\" and now everything seems to work again! The auto-learning package is now included with the rest of the journal in the "autolearnrelease1" folder. Significant modifications will be given new folders.

It is decided that Hassan's server will no longer be utilized in the learning process as it is not reliable enough. Gdansk will have a folder "uploads" which will work as anonymous ftp. This also solves the password protection concern (I may also delete the ftp script when it is finished).

I now create the instructions file for the training software, as seen below.

Instructions for Chess Learning Automation Software

STOP! Before continuing, please ensure that you have the .NET Framework installed on your PC. If you do not, please download and install this from:

**http://msdn.microsoft.com/netframework/downloads/framework1_1/**

1) Go to: http://cegt201.bradley.edu/projects/proj2005/nnchess/.
2) Click on "Download Automated Training Client"
3) Download and Un-zip this file. Do not change anything!
4) Place the folder under the C drive (root). It must go here or it will not work correctly. The folder's path should be C:\autolearnrelease\.
5) Learning requires several programs and scripts to run. Many of these require internet access. If you have a firewall or application/script blocking software, please double click the batch file which is called "StartTraining.bat." Make sure you allow access for each program which runs. Set the "remember box" if relevant (as in ZoneAlarm's case). The .bat file you need to run is located within the autotrainrelease folder.
6) Please ensure your computer is "unlocked" and will stay unlocked. It is also required to disable any screensavers. Please turn off your monitor instead. The screensaver currently confuses the automation script, but this problem may be solved in a future upgrade.
7) Create a new task to run in Windows Task Scheduler. The file to run is "StartTraining.bat." This file is located in the folder which you just dragged to the C drive. Any schedule is fine, but more time is better!

If you need to quit the training process, simply close any open windows. Training will resume from where you stopped it. **Please report any problems you have!** Call me anytime at 847-997-4402 or e-mail jsigan@gmail.com. Thank you for your participation.

The release is almost ready. I need to simply discuss with Dr. Malinowski how the new server set-up will be implemented. A password should no longer be required. The folder contents should also be accessible via http. This will allow the easiest download process later on. A write only folder is created on Gdansk (uploads) which will store all of the trained networks. I will be modifying the FTP script shortly to deal with this change.

The complete autolearning package is now burned to a CD and is installed on a new PC. The computer is an Athlon 64 3500+ with 1GB of PC3200 RAM and a 250GB SATA hard disk. I expect the training speed on this system to be significantly faster than the older P4 3.06 GHz system, and am hoping it can process a network in around 5 minutes. I modify the "configuration.dat" file to contain every network with a name starting in "0." The finished.dat file is emptied. When training starts, I do not notice any errors. SNNS loads in about 3 seconds, while this same task takes about 20 on the older PC. All datasets are downloaded without incident, and the training file is assembled for 0-1.pat without any problems. I have set the training time to 3000 (which is 5 minutes) in the automation script. The first network trains and is uploaded to the server, and then 0-2 starts the process. Again, no problems are encountered. However, when the third network starts training, there is a problem with the save routine! I notice that the automation script does not press the "ok" button as seen in figure 43.

Figure 43: Save dialogue box, showing "ok" button which is not being activated
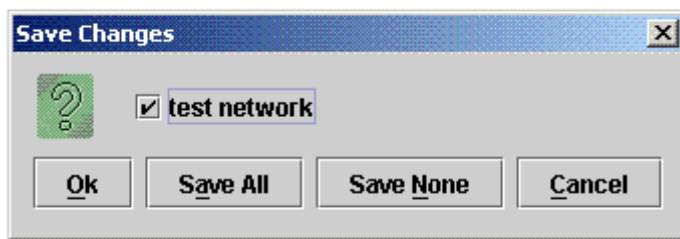
The problem is clearly timing related from my other experiences with this scripting language, so I modify the portion of the script which interacts with the save dialogue. The changes I make are shown in figure 44.

Figure 44: Changes to the script which interacts with the save dialogue box

```
activate,JavaNNS
delay,10
send,JavaNNS,^q
delay,150
activate,Save Changes
delay,40
send,Save Changes,\s
delay,40
send,Save Changes,\t
delay,40
send,Save Changes,\s
```

Delays have been placed between each "keystroke" to allow SNNS time to update itself. After this change, I do not notice any further problems. I let the training run into the night.

2-11-05

I wake up and find all networks starting with position 0 (top left of board) trained! Everything is up on the server as well. There is however a slight problem. My screen is showing an error and is asking me to debug. I can see the cause right away. The end of the configuration file was reached, and the training file generator has tried to create a pattern for a "null" network. It has absolutely no impact on the training, but I will probably get some phone calls about it later so I will try to fix it before releasing the autolearning software to the "public." For now it is added to the list of bugs.

The severe bug that I notice is that the networks are being uploaded, but they all have 0 weights! I am not sure why this is happening, but I come across the cause while experimenting with SNNS. I notice that the 'save' dialogue is different if I wait for the training to complete before saving. Currently, the network is being saved while training is still taking place (a time limit is used to control the amount of training). The difference is that the dialogue after complete training has two check boxes: one for the log file and one for the network. The previous dialogue did not include a provision for the network.

I make a change to the autolearn script, which now controls training time based on the number of epochs (pattern presentations). The change is shown in figure 45.

Figure 45: Changes to the AutoScript file, which add a 'for' loop

```
set,counter,50

:Startpoint
send,JavaNNS,\s
delay,150
dec,counter
onzero,counter,done
goto,Startpoint
:done
```

By changing the set training time to a set number of training epochs, I am ensuring that the network is completely trained prior to saving. SNNS will work better with this method. Currently I am allowing 15 seconds for each epoch. I feel this should be more than enough on any PC which will be used in training. If not, there will be problems with synchronization, and corrections will have to be made at this time.

Testing verifies that the software now works as expected.

2-13-04

After running the training application for around 48 hours, I have decided it is probably stable enough to begin to perform actual training—and distribute to volunteers willing to assist with this project. I have decided to create the configuration files manually for now, as only a handful of PCs will actually be involved. I have found that training a network

takes exactly 16:25 (minutes:seconds) plus around 1:30 for saving, initialization and ftp commands. Thus, 18 minutes are required for each network (with feedforward , partially connected architecture using RPROP algorithm). **More details of the initial network topology will follow shortly**. To train all 1792 networks (the most recent and final count of legal moves), 538 CPU hours are needed (on an Athlon64 3500+ w 1GB RAM). One PC running 24/7 could complete the task in less than 23 days. I have one PC available 24 hours per day, plus another which is available for 12 hours per day. 15 days would be required with just these 2 PCs. Shom Bandophadaya has also installed the training program and will run it for 8 hours per day. This brings the training time down to 12 days. I would like to reduce training time to 7 days, which means I need to find 33 CPU hours per day to add to my existing 44. My goal is to simply locate three people able to run the program for 11 hours+ per day. Given that only 6 machines will be involved in training, it makes sense to supply customized training schedules—these are easy to make by copying and pasting from the list of all possible legal moves, which I already have from when the pattern files were moved to a new directory.

At the request of Shom, the training program has been modified again to display a timer and an easy "quit" button. One of the bugs in the original release of the AutoTrain software was that each cmd window had to be closed separately, and then the Task Manager had to be used to kill the script.exe process. The timer application fixes this problem. When the quit button is pressed, the application identifies all running processes with the names "cmd" or "script." It then kills the processes before exiting itself. The timer is simply a "timespan" object which counts down so the user is able to see how much time remains in a particular training task.

2-15-04

I have realized that I included an older version of the ftp script in the last notebook sent to Dr. Malinowski. I have been trying to destroy any copies of this as it contains login information for read and write access to the server. He has been notified and all other copies have been destroyed.

2-17-05

A screenshot of the timer modification to the AutoLearn process is shown in figure 46. The timer will display the countdown so the user may decide whether to continue training or stop training—which is useful if they need to go somewhere with a laptop, etc…

Figure 46: Timer addition in the AutoLearn software
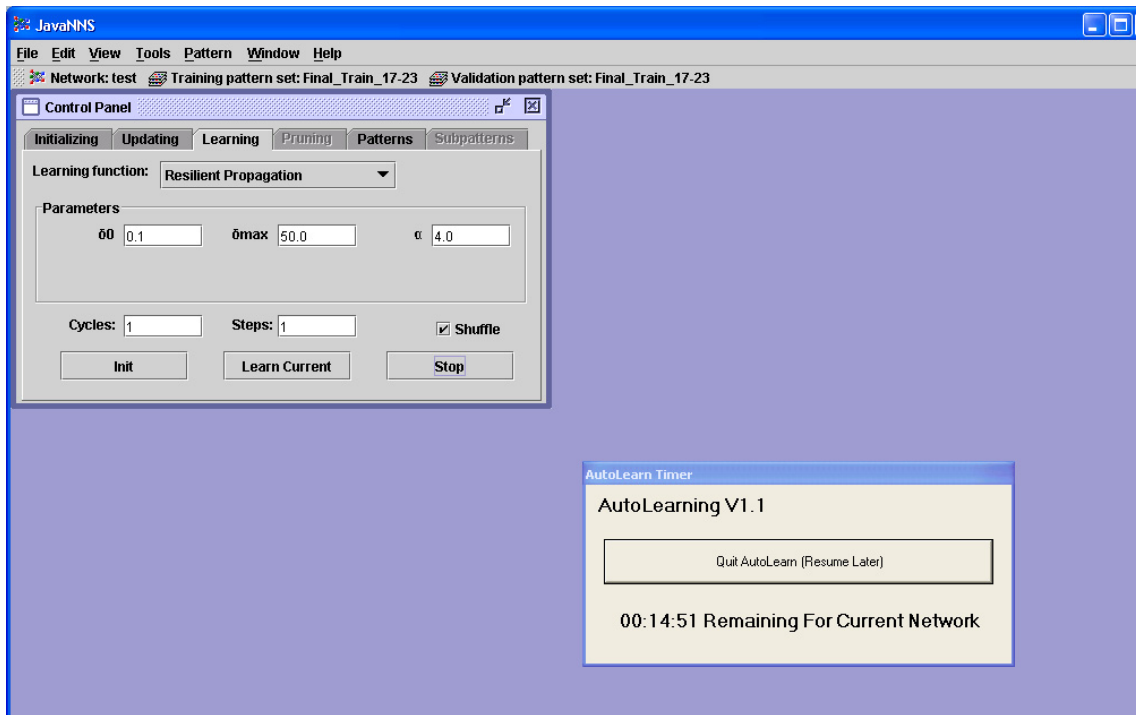
The source code for this timer (and the new quit button) is located in the journal files folder as timer.cs and the application is located in the programs folder as LearnTimer.exe.

Dr. Malinowski has brought up the potential problem with multiple logins to the server. Currently, only two are allowed. With several clients, there is some chance that access could be denied. I will probably want to build in some safeguards to ensure that all data is uploaded correctly if access is denied the first time. For now, it is decided that the number of total logins allowed will be changed to 10.

I have been thinking more about the customized training schedules—it may be better to modify the shell program to choose a random training file and then check to see if it already exists on the server. If not, it will be trained. If it does exist, it will be skipped. The configuration file would no longer be needed. I will look into this shortly, but most likely this will be a version 2 feature (which will train the next generation of networks).

Right now, I will be fixing two problems. First, the ftp script must be deleted when a user quits the application. I add this functionality by:

System.IO.File.Delete("ftp.scr");
Application.Exit();

This is simply placed within the current exit routine inside the LearnTimer application. Next, I fix the trouble mentioned previously that the debugger opens when the configuration file is completely read/finished. I add the same code which executes by pressing the quit button (in the AutoTimer) to the 'if' statement where the "file complete" condition is detected inside the AutoShell application. Now, when the end of the file is reached, all of the training software will be shut down instead of attempting to train a "COMPLETE" network.

A complete version of the most up-to-date training software is located in the autolearnreleaseV1.1 folder. This version is being distributed to Dr. Malinowski, Dan Leach, and Shom B. The training assignments are included on the CDs as:

Me: Start positions 0-31 (60% complete)
Dan: Start positions 32-47
Shom: Start positions 48-55
Dr. Malinowski: Start positions 56-63

Depending on how long it takes each part to train, re-assignments may be made in some cases.

2-20-05

While the networks continue to train, I will make some comments about the current network design and the training settings. The training algorithm being employed is resilient propagation. The algorithm is described in the RProp.pdf document located in the sources folder (see introduction). RProp is a training algorithm which adapts the learning rate **based on the sign** of the derivative of the error curve. The magnitude of the error derivative is not important in determining the learning rate adaptation.

From earlier experimentation with a constant dataset and a single 5 layer X 128 node network, I find that RProp is one of only two algorithms designed for traditional feed-forward networks which will train. QProp (Quick backpropagation) also works. QProp attempts to adapt the learning rate based on current and past values of the error surface slope (derivative). Thus, it is looking at a second derivative calculation. Other algorithms, including the backpropagation algorithm, are observed to produce a flat error surface over all training epochs. Thus, they seem unsuitable for training the given dataset. RProp and QProp both appear to learn, as the error surface in both cases is clearly decreasing over time. Although I do not have a good explanation for this observation, I feel the adaptable learning rate is crucial to the apparent success of these two algorithms. I will be moving forward with the RProp algorithm based on the practical observation that it works, and the evidence suggested in RPropVSQProp.doc in the sources folder. Although the paper concludes that both algorithms are identical in speed with complex problems, it suggests that a slight speed improvement is found with the RProp algorithm in some less complex, or specific problems. Although the study was concerned with character recognition, similarity exists in the size of the input vectors. For the simple case, the paper describes a 5X5 sensor array (25 inputs). The complex case considered 21X21 (441 inputs). My application falls closer to the simple case, as I have 64 inputs. In a casual comparison between RProp and QProp, I find that QProp is actually extremely slow compared to the Rprop algorithm. In fact, it gets stuck after around 10 epochs…I am not sure what is happening (SNNS problem maybe?). Both error surfaces seem to be following the same general shape, so I do not see any benefits to using QProp at this time.

I have nothing to base the learning parameters on, so I decide to leave them at default. If any significant observations are made which contradict this decision, modifications will be made at that time. Now, I decide to examine the impact of changing the size of the network to be trained. I pick a training set (final_train_10-2.pat) at random and create 2 networks. Net1 is a 5 layer network with 128 nodes per layer. Net2 is a 2 layer network with 128 nodes per layer. Each network has 75% connectivity is initialized with random weights -1.0 to 1.0. The training parameters are default, except for the "shuffle" selection. A fairly large training set (9200 patterns) is being used in this test. Figure 47 shows the error curve for Net1 and figure 48 shows the curve for Net2 after 100 epochs.

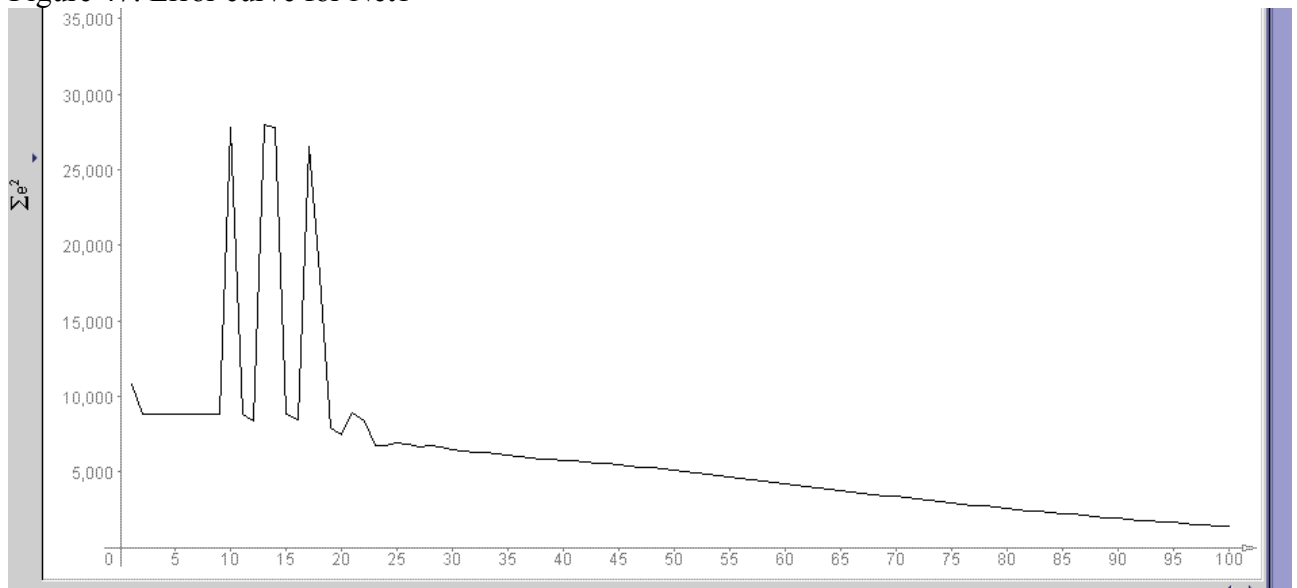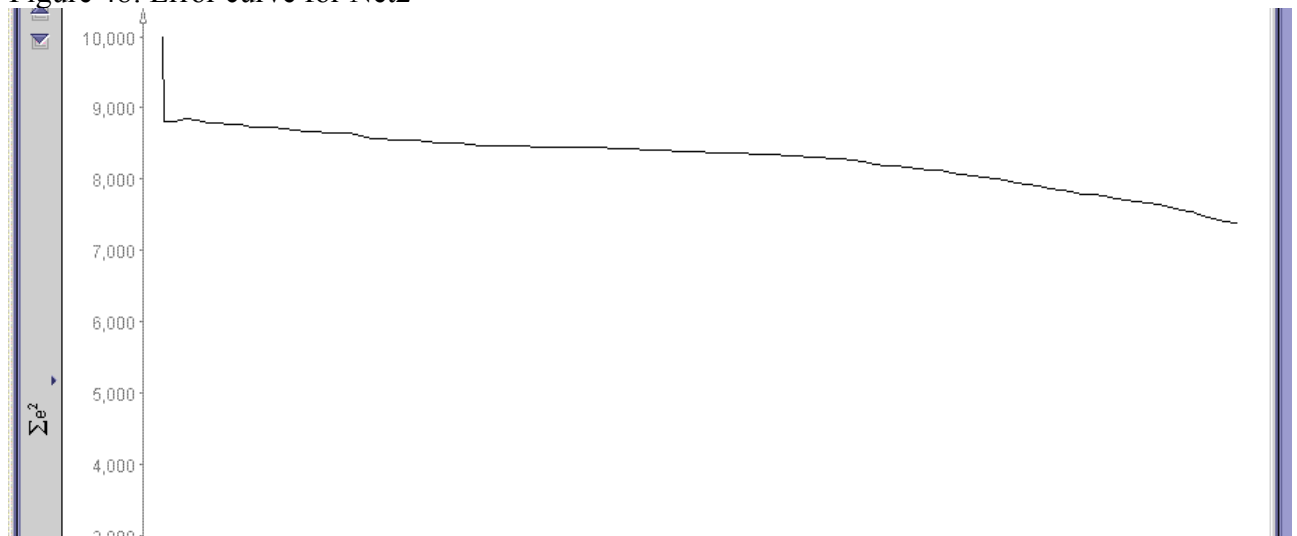Figure 47: Error curve for Net1

Figure 48: Error curve for Net2

The error curve for Net1 shows a high level in instability up until around 25 cycles. This is not observed in Net2. The instability may be due to a learning rate being too high, or it may be caused by "destructive" interference of newly learned patterns on older patterns.

A (higher) learning rate may have more opportunity to impact the eventual output when applied over 5 layers than it does with only 2. From comparison of these plots, it seems that the larger network reaches a lower summed squared error than the smaller network in the same number of epochs. It does take more time to do this however (around 15 minutes as opposed to around 10). I decide to initially start with the smaller network design for several reasons.

- o It is desired to train quickly, and load networks quickly
- o Training must be stable, as there is no mechanism to correct instability
- o Training may take place in multiple stages, so gradual training is desired

From the two plots, it seems that the smaller network converges slightly more slowly, but the speed appears to increase over time. Thus, this characteristic may be helpful if a repeated training cycle is desired. I want to train around 4 networks per hour in order to get the training completed in roughly 1 week on 5 computers, so I will have time for about 50 epochs of training (after allowing for scripting delays to account for the automation challenge). When the 50 epoch mark is compared on both graphs, it seems that the larger network has a slight advantage (7500 SSE) as opposed to the smaller network with around 8500. However, given the much higher stability and predictable nature of the smaller network, this advantage seems less important—what if the instability carried over up to 50 epochs? The data (finished network) could be useless. Another consideration is "overtraining." It is not desired that a network simply memorize all patterns presented to it, but rather create generalizations and learn "schemas." Therefore, it seems more likely that a smaller network would be the best choice, as it lacks the higher memory capacity of the larger network.

Although it is possible that these observations are only specific to this one test case, very little evidence exists outside of it which points to another design. I have chosen to make the hidden layers wider because they may support the development of special relationships better than a narrow hidden layer. This idea is intuitive—as it seems a wider hidden layer would form more combinations from with any given inputs than a narrow layer. The exact number of 128 nodes per layer is arbitrary, and was simply chosen. It may also be possible that the network could be smaller. After the first round of training is complete, assumptions and decisions will be re-evaluated before coming up with architectures for the second series of networks. In summary, the first round of training will be conducted with the following configuration:

Algorithm: Resilient propagation, with default learning parameters
Inputs: 64
Outputs: 1
Hidden layers: 2
Nodes per hidden layer: 128
Training epochs: 50, with shuffled patterns
Initial conditions: Random weights, -1.0 to 1.0
Update order: Serial (Should be fastest for a serially numbered feed-forward design)

Once training is complete, performance will be evaluated, and then the networks will be trained with 50 more cycles to observe the effect. A new architecture will be tried after this.

2-22-05

I have started to develop the interface for the chess playing application. I have thought about the details of implementation, and have decided that writing the entire interface from scratch will be a huge task, and a waste of time for the aim of this project. Therefore I am looking for existing "chess board controls" which are implemented in C#. I find two of them. The first is located at:
 http://www3.sympatico.ca/crchapman/rnchesscontrol_full.html
This control looks very promising, as seen in figure 49.

Figure 49: RNChessControl Screenshot

I download the dll files, but after over an hour of trying, I can not get the control to import. References can be added to the project without any problem, but when trying to update the toolbox, an error occurs and states that some references are missing. The documentation does not help. I will try to recompile the dll files if the second source does not prove to be effective. Perhaps the dlls were compiled in a different version of .NET?

The second chessboard dll which I find is located at:
http://www.codeproject.com/csharp/GpChessControl.asp

This control also looks like it may be useful, as seen in figure 50.

Figure 50: 2nd ChessBoard Control screenshot

Unlike the last control tested, this one is very easy to import and work with!

2-24-05

The chess board control which is shown in figure 50 will be used for the rest of the project. It seems to be exactly what I was looking for. Moves are validated, it generates EPD (FEN) strings, and it is easy to make the computer make a move! It can also check for end game conditions (checkmates). I will be working on an interface design today. Networks are currently around 80% trained, so I should be ready to start building the playing system within the week. The original code and files for the chess board are located within the programs folder, under the ChessBoardctrl folder.

Dr. Malinowski has had some trouble running the training software, but the problem has been found. The server storing the data files was restricted to on-campus locations, so the files were not accessible outside. It has been fixed.

After several hours of experimentation with the chessboard control shown in figure 50, I decide to implement the features shown in figure 51. Note that the control only includes the board (and movement logic), and that other controls/functions must be added manually.

It takes some time to determine how to make a "computer move." After calling the "updateboard" method, with a standard PGN position notation, it is required to call refresh so the board is redrawn. If this is not done, the board will not update until the user makes another move.

The control does include a convenient method for "getFEN" which is extremely useful for this application. It will be called, and the result will be sent to the existing functions for conversion to a "floating point string" (see the EPD_FP application). **The floating point string will be sent to all neural networks once they are trained and incorporated into an evaluation application.** The history field shown in figure 51 will display the processing steps…another field may be added in the future to allow the PGN notation of the game to be displayed.

Figure 51: Current Progress on the playing interface
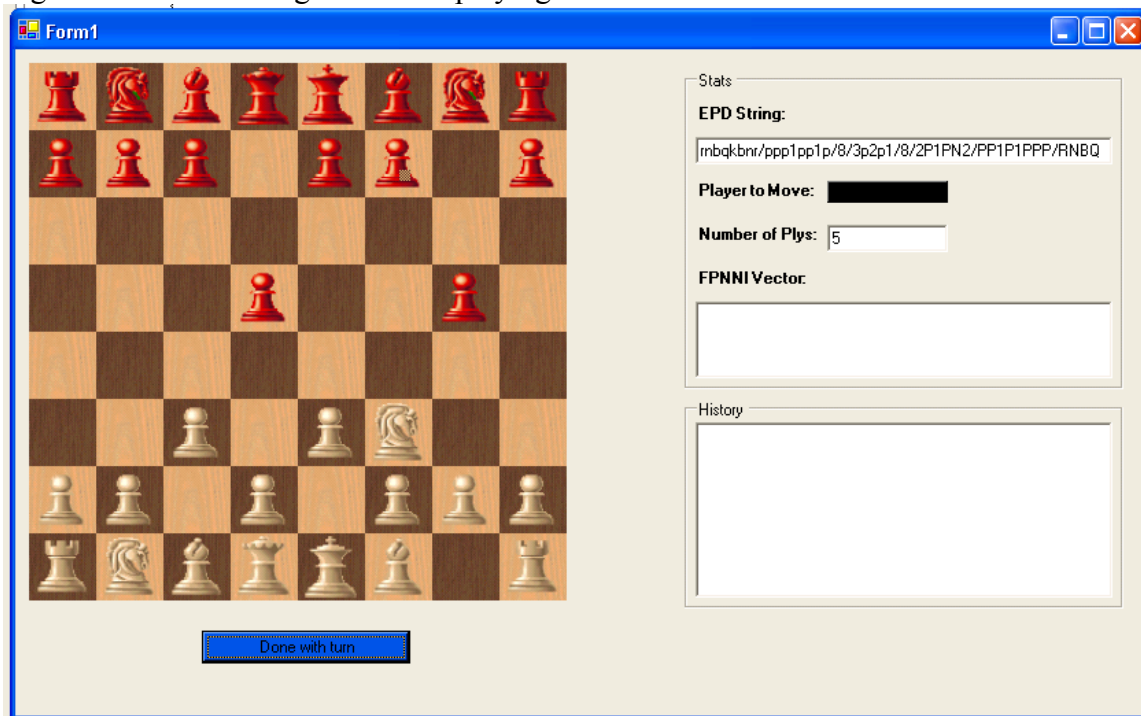
The Floating Point NN Input Vector is still not displaying, despite the fact no errors are being returned. I will be looking into this problem shortly. I suspect a problem with the

format of the FEN string (possibly a syntax issue compared to earlier work with EPD strings). I will check my functions for such things as slash direction (\ or /).

The next step is to create the module responsible for loading a network file and generating an output. Hopefully, this task will prove to be little more than parsing the .net file and performing some matrix multiplications. I do not anticipate a delay here.

I have been considering some alternative structures to the feed-forward networks currently being investigated. My most recent (and maybe promising idea) is to use a LAM approach (linear associative memory).

In order to do this, I am proposing generating a set of "one-hot" keys—each with 1792 elements. 1792 keys would exist in all (forming vector set A). Each pattern in the training set (or as many as possible) will be converted to a vector of 64 floating point values, with zero padding to make 1792 total elements in each vector. 1792 vectors would be chosen at random from the input vector space to form set B (one from each move collection). A and B are both composed of column vectors. Next, memory matrix M would be created, as expected, by $M = \sum_{i=0}^{1791} B_i A_i^T$. However, at this point something must be changed. In LAMs, a pattern "a" would be presented and some response "b" would be obtained. It is required to present "b" in this case and get "a" as a result…I think this can be done, but I will need to look at it some more over the weekend to be sure.

Assuming this idea works, and a key can be returned from a "corrupt pattern" instead or the typical "corrupt pattern" returning a key, I will generate as many memory matrices as possible. Ideally, all known patterns would be represented. While playing, the board position would be presented to all memory matrices, and responses would be gathered from each memory matrix that indicate which key is the closest match to the output. A system of voting would be created, which I feel could have enormous power in making the move decisions.

It was not included previously, but the code for the interface up to this point may be found in the journal files folder as interfaceCodeV1.cs.

3-1-05

The problem with the NN input vector has been fixed. I found that the string did not end in a space, which was the case in the original function's use in the data processing. I insert a space at the end of the FEN string: `epd.Text=chessBoard1.getFEN()+" ";`, and the problem is fixed.

The problem with the LAM can be solved fairly easily, although the actual implementation may be very time consuming (depending on the efficiency of a matrix inverse algorithm). In order to get a key A as an output when a pattern B is presented to M, it is required to produce the inverse of M, or M^-1.

M-^1 will be taken on a 1792x1792 square matrix, so it may be an extremely long calculation. I will be looking for pre-existing functions to perform this task in C#, as the algorithm could be very complex and will take a significant amount of time to write from scratch. I will also need to decide if it actually makes sense to make a difference calculation between a board position and a known vector...is the norm^2 error meaningful?

I am now trying to figure out exactly how SNNS computes the output for a network. Each node in the network has a "bias" term. I do not understand exactly how this is being applied (+,-?), so I look in Google for example calculations or even an existing application which will load the file and perform the task.

I find a good reference (see "Representing Bias in SNNS" located in the sources folder). Although SNNS represents a bias for the input units, **it does nothing with them**. Now it makes much more sense. Bias is basically a threshold value. In practice, it will simply be added to the summed inputs.

From this reference, figure 52 is obtained which describes the calculation process for each node.

Figure 52: SNNS node output calculation
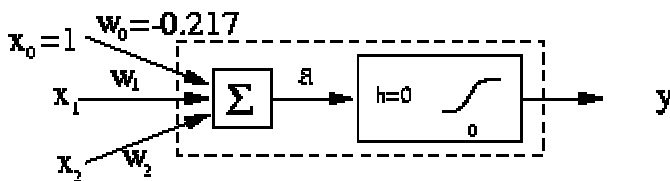
Of course, the activation function in my case is the hyperbolic tangent. The built in function will be used in C# (Math.tanh).

I am now working on creating the function which will read the completed .net file and parse the content (and fill various arrays with the data) in order to produce an output value. The flowchart shown in figure 53 describes the overall functionality. Of course, the solution side (on the right) will require an input vector, I(n) which is the board position an some point "n" in the game. I will create an individual function for the loading phase and the solving phase. A new class is defined for "network" which will have as public properties arrays for bias, weights, and activations (outputs). The methods load and solve will be created.

Figure 53: High level flowchart for the network parser and solver
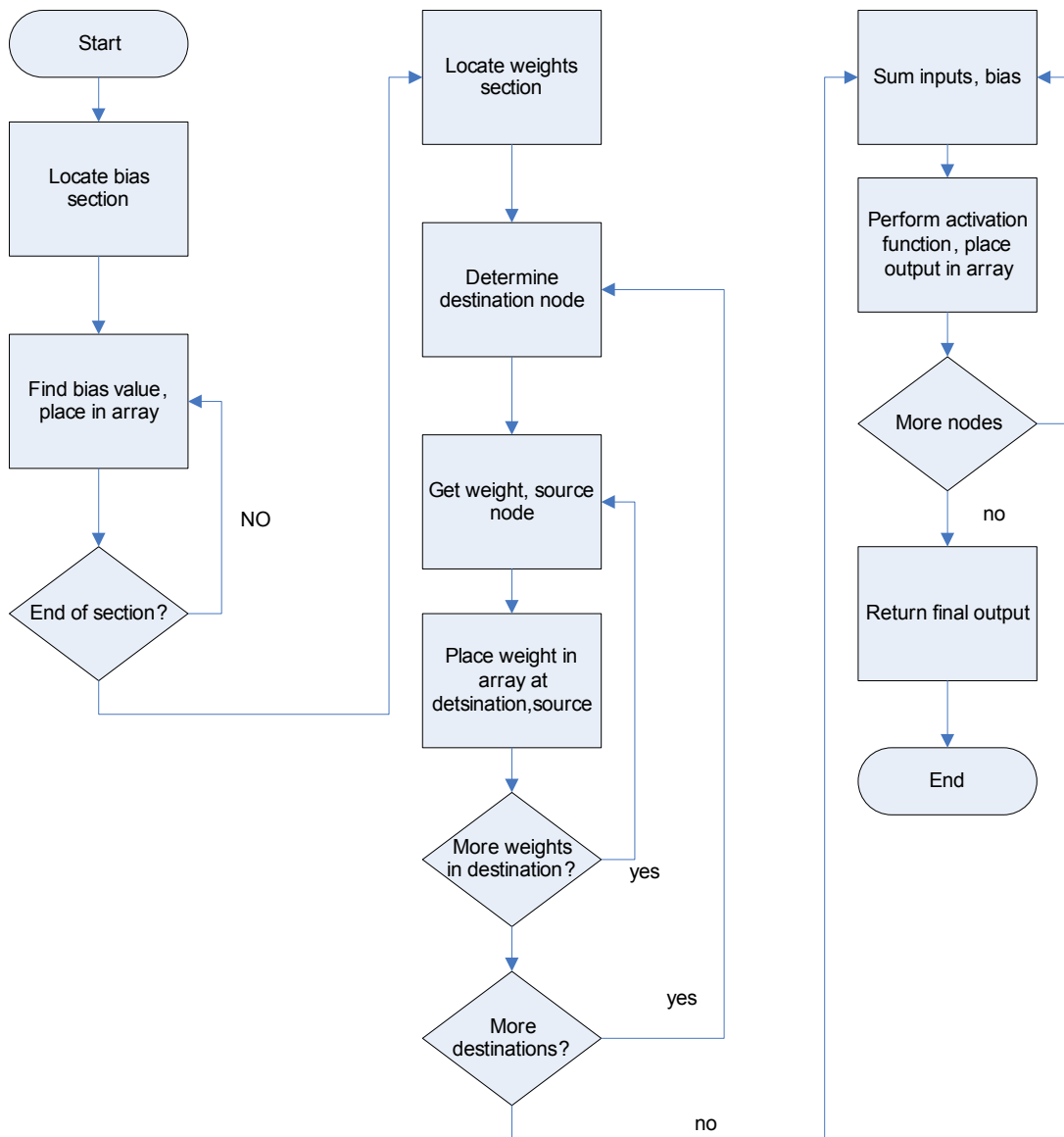
In creating the class, no significant technical difficulties are encountered other than the common file navigation challenges with parsing any file. The solve method proves to be slightly more difficult. I find a major source of difficulty in thinking about the problem is that SNNS uses a "1 reference point" in its network files, but C# uses 0 references. I decide to make all arrays one position larger than the networks and simply not use the 0 position. This explains the somewhat strange loops which start at 1 in the solve method. The code is much easier to follow with this change.

At first I think it may be required to pass through the solve routine numerous times until an output is stabilized (as in a recurrent network). However, the network is arranged in serial order, so connections to a specific node must originate in nodes with a lower ID. Thus, only one pass is required. It seems somewhat beneficial to include this code here,

so it is shown in figure 54. I confirm it works by presenting a few patterns, and then presenting the same patterns in SNNS. Both solutions match.

Figure 54: Network solution method (after arrays are loaded with parsed net file)

```
public double solve(double[] board_position)
{
        for (int count=1;count<65;count++)
        {
                output_activations[count]=board_position[count-1];
        }

for (int destination_node=65;destination_node<322;destination_node++)
{
        for (int source_node=1;source_node<322;source_node++)
        {
        input_sums[destination_node]=input_sums[destination_node]+output_
        activations[source_node]*weight_array[source_node,destination_nod
        e];
        }
        input_sums[destination_node]=input_sums[destination_node]+bias_ar
        ray[destination_node];

        output_activations[destination_node]=Math.Tanh(input_sums[destina
        tion_node]);
}

        //MessageBox.Show(output_activations[320].ToString());

return output_activations[321];
}
```

The code for the entire class is found in the journal files folder as network.cs.

3-3-05

The goal for today is to get the system to start playing on its own. All components now exist. My only remaining concern at this time is that the networks may still try to make illegal moves. Although the chessboard control does not allow an illegal move to be made by dragging and dropping a piece, I find that illegal moves can be made when the "updateboard" method is used—if is possible to move knights horizontally for example.

I check the control for any methods to check the legality of moves, but I am not able to find any at this point. Therefore, I decide that I will have to create further logic to determine if a given move is legal given a particular board position and piece. I decide to break this task up according to piece. Thus, the required rule logic will check for pawn, bishop, knight, rook, king, and queen moves. Queen moves will be a set of bishop and rook moves, while the king check will be a limited version of the queen check (with a move radius of only one square).

The logic for each move checks to see if the start position is held by the particular piece in question and that the final position is either empty or held by an enemy. In the case of the rook, bishop, and queen, the path between the start and final position is checked to ensure a free path exists. Pawns check for a free path only if the start position is on the original position (2nd row to the top). Kings do not need a free path check. An example of the legality checking code is shown in figure 55.

Figure 55: Checking move legality for the rook

```
static bool check_rook(Position start, Position end, double[] board)
{
bool legal=false;
int count=Math.Max(Math.Abs(start.x-end.x),Math.Abs(start.y-end.y));

if (board[end.linear_represenation]<=0)//could bel legal
{
        if ((start.x<end.x)&&(start.y==end.y))//left to right
        {
                legal=true;
                for (int z=1;z<count;z++)
                {
                        if (board[start.linear_represenation+z]!=0)
                        {
                                legal=false;
                        }
                }
        }
        if ((start.x==end.x)&&(start.y>end.y))//bottom to top
        {
                legal=true;
                for (int z=1;z<count;z++)
                {
                        if (board[start.linear_represenation+z*(-8)]!=0)
                        {
                                legal=false;
                        }
                }
        }
        if ((start.x>end.x)&&(start.y==end.y))//right to left
        {
                legal=true;
                for (int z=1;z<count;z++)
                {
                        if (board[start.linear_represenation-z]!=0)
                        {
                                legal=false;
                        }
                }
        }
        if ((start.x==end.x)&&(start.y<end.y))//top to bottom
        {
                legal=true;
                for (int z=1;z<count;z++)
                {
                        if (board[start.linear_represenation+z*(8)]!=0)
```

```
                    {
                            legal=false;
                    }
                }
            }
    }
    return legal;
    }
```

A similar check is created for each remaining piece. The legality checks are to be inserted into the interface code prior to evaluating a given network's response—this will speed up the process by eliminating evaluation of illegal moves.

The main legality check function (composed of a call to each piece check) will simply check to see if any of the piece-specific move choices are legal for a given input of start and end positions **and** board position. If so, a true is returned. The code is integrated with the interface code, and can be seen in the journal files folder as interfaceV2.cs. Now, every move returned by the system should be legal.

Now, when testing the code with a MessageBox giving the outputs for various board inputs, I notice that several networks give saturated outputs (1 or -1). Although this may be fine, I am worried that the results may not reliable in these cases. ~~Thus, I decide to choose the network response which is the highest legal response BELOW 1.0. I will look into this more later, but it should work for now, at least to prevent a single network from always winning the vote.~~

For now, I decide to allow the saturated outputs to be returned. It is probably best to start with a list of returned moves (top 5 highest). Getting the interface window to work is surprisingly easy…the networks are simply placed in the same folder as the executable, and the networks are loaded one by one (if legal move==true) (see page 59). After evaluation, the "best output" is updated if the current output is higher. It is possible to play the game now! However, the performance is less than desired…in "playing" (moving pieces simply to observe responses), I notice a few things which must be fixed right away.

Several move errors are corrected as they occur. The system has tried to make the following illegal moves:
Pawn captures vertically
Pawn moves diagonally
Rooks move through pieces

The problems are simply logic errors in the rule checks…such as forgetting to check for rook moves when evaluating a position or not checking the last position in a free path check.

While playing, I also notice that several moves do not get made, even though at the time they are clearly the best moves to make.

The knights NEVER move!
I also notice that the king moves are always seen as illegal (and the king will not move).

I will need to look into this more closely, as the cause is not as obvious as the previous logic errors listed above. I am also looking at how to make the decision to make a move by evaluating the 5 "top" moves…is there something which can be added?

3-5-05

I need to re-examine the logic used to determine legal moves, as there is clearly something wrong with it based on the above errors. I find that the king, being a subset of queen moves (with move radius equal to one) is not moving because it fails the identity check (which checks for .9 in the start position). I remove the check (it is not really needed, as another function calls the individual legality checks and can check for a queen at that time). After re-compiling, I find the king will move (albeit not always intelligently). I will be closely examining the specific moves shortly, and they are being ignored for now as I want to ensure all rule logic is correct.

Once every piece is ABLE to move, I find the NN system is definitely going to need additional help in deciding a proper move. The system seems to return some intelligent moves in the list, but the output values are very close to one another, and I doubt the differences have any significance. The maximum value is not always the best.

The code for the legality checks is becoming very confusing, and I think it would be better to **create another class for actually solving the move** (including the possibility of obtaining scores for specific features (will be described shortly). The class is called "Rules_Evals." It will be created within the next couple of days.

I need to start working on the presentation to be given on Tuesday, which is a progress report of the project.

3-7-05

For the presentation, I would like to discuss the topics:

- Project background
  - NN system design
  - "Process" design
- Schedule and timeline
- Overview of progress
  - Training discussion
- Recent results and findings
- Current direction and goals
  - Discussion of evaluation functions

The "evaluation function" is the method proposed to help make decisions when a list of several moves is returned by the NN system. Although evaluation functions are doubtlessly extremely complex in certain commercial chess programs, I feel that very simple "game characteristics" could be used to add significant playing ability to this system. In particular, it is desired to give the system a sense of four key areas:

Material (ΔM)
Threats (ΔT)
Mobility (ΔO)
Vulnerabilities (ΔV)

These are intuitive concepts, which can be used to determine the relative strength of any move **given a change in the parameters from the current board position**. In short, a relative score can be assigned to a move, and the move having the highest score in the list will be made (as in figure 56). However, I can see two possible ways this score could be found. The first would consider the above four properties alone (maybe adding more if need be) for each of the top network outputs. The network output itself would not be used in the score derivation.

In the second method, the network output would actually be used in the score calculations. Therefore, the equation would be Score = $a*\Delta M + b*\Delta T + c*\Delta O + d*\Delta V$ for the first method and Score = $a*\Delta M + b*\Delta T + c*\Delta O + d*\Delta V + Y$ for the second. The values a,b,c, and d are scaling factors which will be determined largely by trial and error.

Figure 56: The application of the evaluation function—which move is best?

Code will be completed for these initial 4 evaluation function parameters. It will probably be integrated into the Rules_Evals class once it is completed. For the sake of clarity and completeness, Dr. Malinowski has suggested that slides be added which describe the training process. An entire flowchart of the training process actually does not yet exist, so figure 57 shows what is happening.

Figure 57: Complete flowchart of the training process
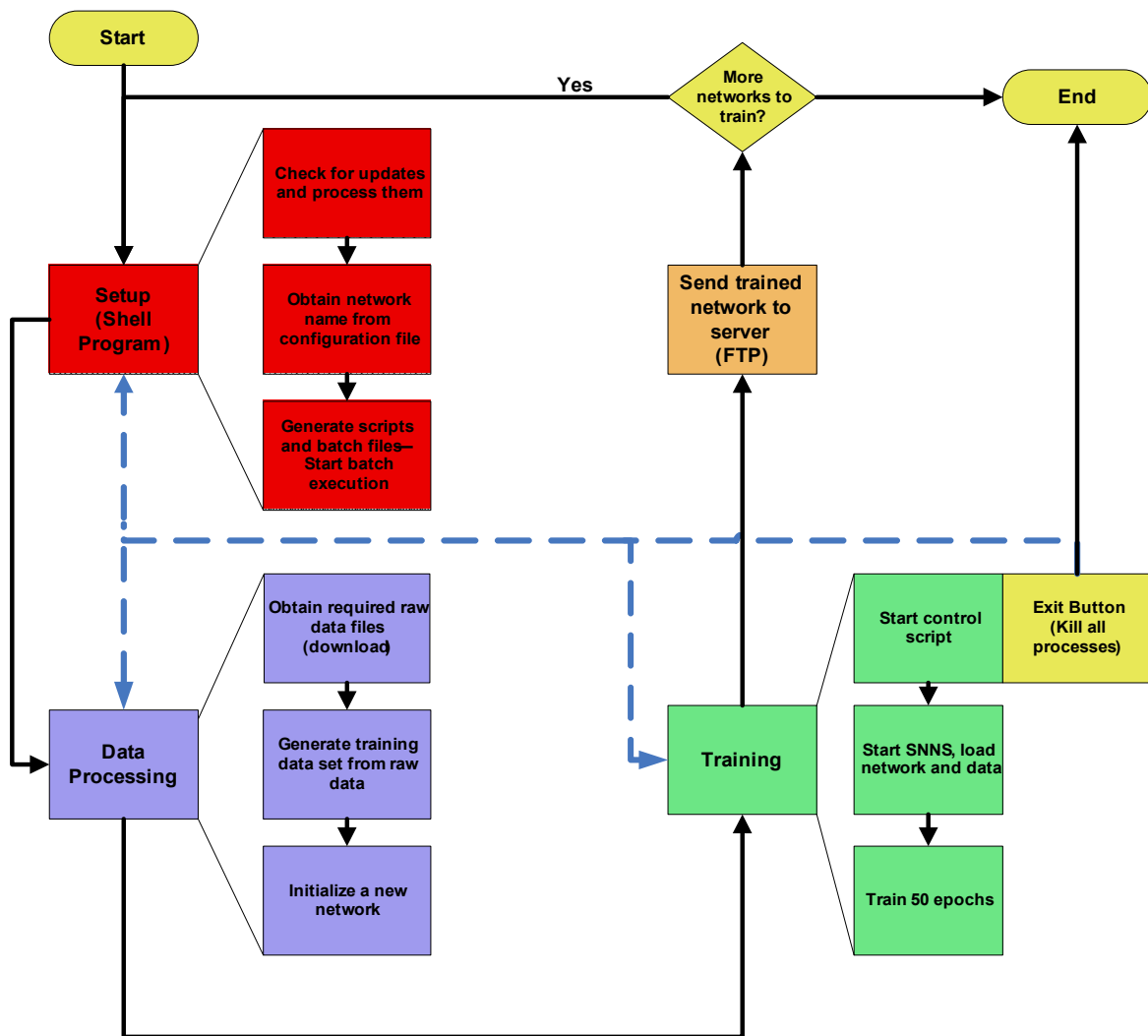
Finally, the presentation calls for an overview of the current schedule. After considering the work which remains to be done and the due dates for the various deliverables, the schedule in figure 58 is completed.

Figure 58: Updated schedule

Date	Goals and progress
8 March-05	Progress presentation
3 March to 17 March-05	Develop evaluation function / enable automatic movement
10 March to 19 March-05	Train networks (round 2)
10 March to 24 March-05	Look into LAM approach (and create memory matrices)
10 March to 1 April-05	Compare performance (random moves, computer players)
20 March to 27 March-05	Tweak evaluation function
31 March to April 21-05	Prepare for presentation and expo
April 21-05	Expo
April 28-05	Presentation
May 9-05	Final report (draft due 4-15-05)

Although effort will be made to follow this, changes may still occur.

The completed version of the presentation can be found in the journal files folder as "progress.ppt."

3-8-05

Aside from presenting the progress report, a major goal for today is to complete the Rules_Evals class. This class will contain a constructor taking a board position, start, and end location (of type Position). The board will be a 64 element vector. The constructor will determine whether the move is legal or not, and will also evaluate any evaluation function parameters. The legality check is exactly the same as the original.

Some major changes do have to be made however. All methods of this new class which are involved in the legality checking must be changed so that they do not take input parameters (they will simply access public properties of the object). Thus, several public variables are declared as members of this new class:

```
public double[] BOARD;
public Position START;
public Position END;
public bool move_is_legal;
public double material_score;
public double vulnerability_score;
public double threat_score;
public int mobility_score;
```

```
        public double composite_score;
```

The variables used within each "piece check" must be changed to match these new variables (they were lowercase before). This takes some time to complete, but once it is finished the class seems ready. I compile everything, and the game runs. At this point, the class does not contain anything additional to the original rule logic. Thus, it is possible to read the "move_is_legal" property, but no further value is added. The main project code is cleaned up significantly however.

Now, when determining the best moves, it is possible to work with a single vector of Rules_Evals objects. This is much simpler than the previous method. I now add the function to check for the material score. This method is very simple, and simply sums the value of all pieces on the current board. The value of all pieces is also summed for the board which would exist if the move was made. The difference between the values is determined and is the end result of the calculation. Figure 59 shows this method.

Figure 59: Deriving the material score

```
public double derive_material_score()
{
double[] newboard=(double[])BOARD.Clone();

newboard[END.linear_represenation]=BOARD[START.linear_represenation];
newboard[START.linear_represenation]=0;

        double change=0;
        double new_sum=0;
        double old_sum=0;

        for (int i=0;i<64;i++)
        {
            new_sum=new_sum+newboard[i];
        }
        for (int t=0;t<64;t++)
        {
            old_sum=old_sum+BOARD[t];
        }

        change=new_sum-old_sum;
        //MessageBox.Show(change.ToString());
        return change;
}
```

The code for Rules_Evals (up to this point) can be found in the journal files folder as Rules_Evals _050308.cs. The vulnerability check is also started, but it is not finished yet.

3-10-05

The vulnerability score is going to be a central focus of today's work. A few decisions must also be made regarding the second round of training, and how this will be

implemented. Dr. Schertz has offered to run training on his PC, so I would like to get him a copy of the training software if possible. He will need data on a CD however, as his internet connection is too slow at home. This may be problematic after thinking about it (due to time constraints and space on the CD), but I will see if there is a solution. It may also be possible to simply train three of my computers over spring break without the need to divide training further—the second round will not need as long as the networks do not have to be initialized this time.

The training software will have to be modified to perform the second training round. First, all data on the server will be placed in another directory as backup. There are a few networks which may not be on the server, as they were trained but not uploaded. I have a full set on my PC, so this will have to be transferred to the server before any training can take place. The change to the training software should not be too difficult—it is now required to download a specific network file instead of making a new one. The dataset generation is the same. However, the script which runs SNNS will have to have some heavy modifications. The FTP process is not any different (we will use the same directory on Gdansk once the data has been backed up).

For now, I am continuing to work on the vulnerability score derivation. Part of this code may already be seen in the Rules_Evals _050308.cs file. The procedure for finding the vulnerability score for the board involves locating each black piece and determining whether or not it is threatened by a white piece. If it is, then the value of the black piece is added to the running score. To get the vulnerability score, the old score is subtracted from the new score. A negative result is better.

In order to check an individual piece for vulnerability, the white enemy pieces are converted to black, and the black piece is converted to white to perform movement checks. This is required because the movement checks only work for black's move. Thus, the goal is to determine if the black pieces (which used to be white) can legally move to the square occupied by the "target." If a move exists, we know the piece is vulnerable.

After working the entire morning and part of the afternoon on trying to get the function to work, I am not having any luck. The function is always returning a score of 0 for the vulnerability! The code at this point may be found in the journal files folder as Rules_EvalsV2_050310.cs.

Talking to Dr. Malinowski helps a great deal in solving this problem. At first, it is thought that the failure may be caused by the use of floating point (actually double) in the input vector. If any values are used in a calculation (such as multiplying by -1), it is possible that the value may not be exact. Some error may be introduced (.1 may be .09999998 for example). If this happens, "if statements" which check for the value will fail. However, after changing the argument of "if statements" to a range, no improvement is observed. Something else is the cause.

 The logic of the vulnerability function works in most cases, **but will not work in the pawn case**! Figure 60 shows the board I am using for the test.

Figure 60: Testing board position for the vulnerability score function

The reason the test is failing to produce the proper output is that the black pawns (red) can only move toward the bottom! Therefore, when the board is "switched" for the calculation, the black pawns (formally white) would not be allowed to move towards the top. Therefore, no vulnerabilities are detected. This idea is confirmed by the board position shown in figure 61.
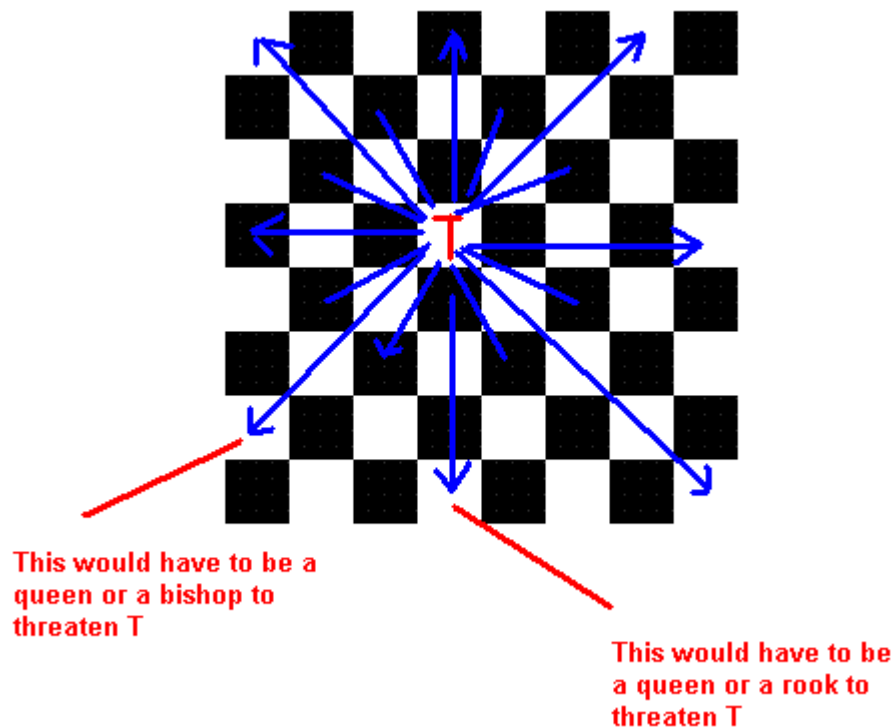
Figure 61: Confirmation of the cause of vulnerability check failure
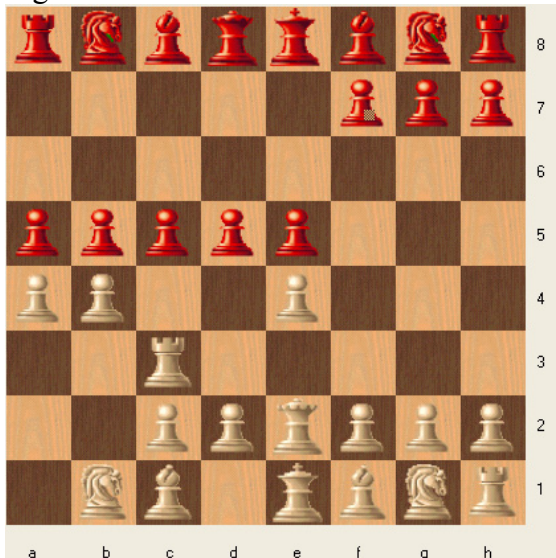
When this board is presented, the current vulnerability score (not including the future position is given as 0.2. This answer is correct, as the white rook on c3 threatens c5 and e2 threatens b5. Both targets are pawns with .1 value. Now that the problem has been found, the solution can be made in one of two ways.

The first way is to "flip the board" when the pieces are inverted. This would be more difficult and much slower than the second option, which is proposed by Dr. Malinowski. The proposal would move out from the target piece in all directions, and check if each square contained a specific piece which would be allowed to attack the target. Figure 62 explains this better.

Figure 62: Checking for piece vulnerability, method 2

This would have to be a queen or a bishop to threaten T

This would have to be a queen or a rook to threaten T

3-15-05

Work has started on rewriting the vulnerability check as described in figure 62. The function consists of several "if" statements, which determine the direction of piece travel. Once travel direction has been identified, the positions are checked in a direction **AWAY** from the starting position. If the square contains a piece which could attack the start, then the start location is considered vulnerable. If the square contains a piece which is not capable of attacking the square, the direction of travel is considered blocked, and further checking in the given direction is stopped. The code for checking the up direction of travel is shown in figure 63.

This same technique is used for checking all other directions of travel. In the other cases, it is required to change the argument of the while statements, and the direction of checking. For example, figure 64 shows the check for the up/right direction.

Figure 63: Code for checking the up direction of travel for vulnerabilities

```
//check up
bool blocked1=false;
index=input_target.linear_represenation-8;
while ((index>=0)&&(blocked1==false)&&(threatened==false))
{
        if (((input_board[index]==-.5)||(input_board[index]==-.9))&&(blocked1==false))
        {
                threatened=true;
        }
        else if ((input_board[index]==-1.0)&&(index==input_target.linear_represenation-
8)&&(blocked1==false))
        {
                threatened=true;
        }
        else if (input_board[index]!=0)
        {
                blocked1=true;
        }
                index=index-8;
}
//MessageBox.Show("Up ok");
```

Figure 64: Code for checking the up/right direction of travel for vulnerabilities

```
//check right/up_diagonal
bool blocked5=false;
index=Math.Min(input_target.y,(7-input_target.x));
YF=input_target.linear_represenation;//offset (original index)
int count=1;
while ((count<=index)&&(blocked5==false))
{
        if (((input_board[YF-(count*7)]==-.3)||(input_board[YF
(count*7)]==-.9))&&(blocked5==false))
        {
                threatened=true;
        }
        else if (((input_board[YF-(count*7)]==-1.0)||(input_board[YF-(count*7)]==-
.1))&&(count==1))
        {
                threatened=true;
        }
        else if (input_board[YF-(7*count)]!=0)
        {
                blocked5=true;
        }
        count++;
}
//MessageBox.Show("RU ok");
```

For an initial test of the function, I leave out the knights. I find the check causes an error,
and quickly find that several "while" statements have condition checks with the incorrect
operator…< is mixed up with > in some cases…for example. This error is causing invalid
indexes to be "checked" in arrays (piece checks), which is causing the program to crash.
The code is checked for these problems, and now appears to be fine when it runs in the
evaluation function.

The knight check is now added—which simply checks specific squares (using constant offsets from the start position) for the enemy knight. Figure 65 shows how this is done.

Figure 65: Code checking for vulnerability to enemy knights.

```
//Check knights
int T=input_target.linear_represenation;

if
((((input_target.x-1)>=0)&&((input_target.y-2)>=0)&&(input_board[T-17]==-.4))||
(((input_target.x-1)>=0)&&((input_target.y+2)<=7)&&(input_board[T+15]==-.4))||
(((input_target.x+1)<=7)&&((input_target.y-2)>=0)&&(input_board[T-15]==-.4))||
(((input_target.x+1)<=7)&&((input_target.y+2)<=7)&&(input_board[T+17]==-.4))||
(((input_target.x-2)>=0)&&((input_target.y-1)>=0)&&(input_board[T-10]==-.4))||
(((input_target.x-2)>=0)&&((input_target.y+1)<=7)&&(input_board[T+6]==-.4))||
(((input_target.x+2)<=7)&&((input_target.y-1)>=0)&&(input_board[T-6]==-.4))||
(((input_target.x+2)<=7)&&((input_target.y+1)<=7)&&(input_board[T+10]==-.4)))

{
      vulnerable=true;
}
```

3-17-05

Continuing with the evaluation function development, I will now complete the "threat" check. This check will simply check the value of enemy pieces which are currently threatened by the black side. It is essentially the exact same function as the vulnerability check created previously.

To complete this function, I take the vulnerability check function from above and change the signs on all pieces that are checked for. It is important that the check for pawns will take place in the DOWN diagonals in this case (they take place in the up diagonals for the black side's vulnerability check).

The implementation of this function is fairly straightforward, and no errors aside from normal typos are encountered. However, there is still not any verification that the function actually works…this will have to be tested by printing the scores to the screen for each component of the evaluation function once all parts have been implemented. The reported scores will be compared against the board and checked. Mismatches will have to be dealt with on a case by case basis.

The code file for the newest evaluation checks is found in the journal files folder as Rules_EvalsV3_050317.cs. Also see the file InterfaceV3_050317.cs for the most up to date version of the interface…which now displays evaluation components.

3-19-05

Although the mobility score has not yet been dealt with, I decide to attempt playing the system up to this point—as a rough estimate of performance. The game will not be recorded for this time, as I simply want to know if the EF is making any sense. For the

first test, I decide to take the top ten moves based on neural network output. These moves will have evaluations calculated by:

```
composite_score=material_score-vulnerability_score+threat_score;
```

Notice that the vulnerability score is negative. This is because the sum of vulnerable pieces is desired to be low. Thus, the ideal direction of travel is negative (new-old).

The performance is very disappointing. The system manages to lose in only 8 moves, and appears to be playing little, if any, better than random piece movements would allow. Given this initial result, I do not feel that the current EF and method warrants any further testing. However, this test does verify that the components of the evaluation function are returning values—and they seem to be in the proper direction. Verifying the exact magnitudes is proving to be extremely time consuming by hand. However, in the first few moves this is manageable, and all components appear to be working.

How can performance be improved?

3-20-05

I decide to check the performance using a new form of the evaluation function. The new form will take into account the magnitude of the neural network output in forming a result. Still, only the top ten neural network outputs are considered as contenders. The new function is:

```
composite_score=material_score-vulnerability_score+(net_eval*.5)-
.2*threat_score;
```

The sign on the threat score has been changed, as the function is giving old-new (and higher threat scores are "better"). Thus, the better score is negative. This error (see previous evaluation equation) causes me to try the previous test again. However I see no difference in the overall quality of play.

Now, the new function is used. The weights, (.5 and .2) are just guesses, and are based on nothing but the fact that I am seeking some kind of balance between offensive and defensive play. From the previous games, I have noticed that the computer is trying to use high valued pieces to set up attacks on enemy pawns, only to place itself in position for attack by other pawns or pieces, for which it has no defense. This behavior would seem to indicate the threat score is being given too much value, and the vulnerability score (more defensive in nature) is not given enough.

In playing this new game, I feel there is a great deal of improvement. The computer has managed to survive for much longer (40 plys) and has managed to capture several of my pieces. However, it does eventually get in a hopeless position, as seen in figure 66.

Figure 66: Final position when using the improved evaluation function
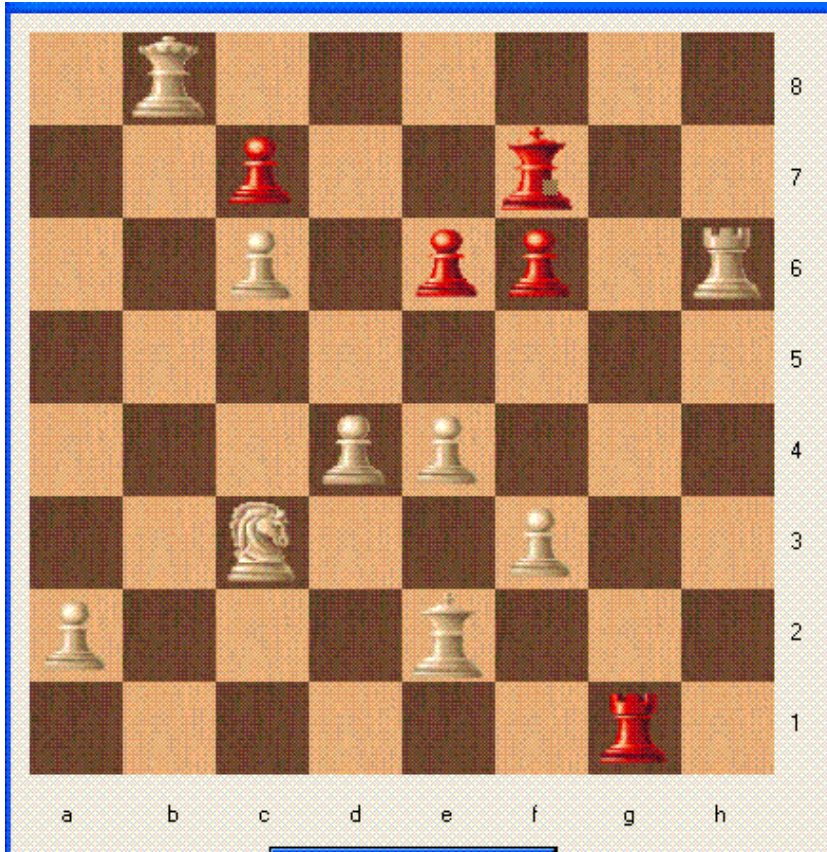
In this case, it seems the computer has surely lost, however I do not put it in check or checkmate just yet. I see the red king moving between it present location and e7. Once in e7 it goes back to f7. This behavior repeats over and over when I move pieces not seeming to be relevant to the attack setup (my king back and forth for example). Clearly, this is not the proper thing to do. Much more could probably be gained by moving g1 to g8 for example.

I play a few more games, and several times notice an "oscillation" which is in all likelihood detrimental to performance. In most cases, the system moves the knights b8 and g8 to c6 and f6. Then, it repeatedly moves a8 to b8 and b8 to a8 repeatedly! This is a ridiculous move! I think it is occurring when no returned moves provide a gain in terms of evaluation score. It seems that two networks have been trained (b8-a8 and a8-b8) which just happen to produce the highest outputs in alternating orders…which is a strange situation indeed. The rightmost rook will also do this! Therefore, the situation is not isolated, but I am not sure exactly what implications it has at this time.

At this time, I am somewhat pleased with the progress, but also have a few concerns. Although there seems to be *some* promise in the neural networks, the current solution is still far from "optimum." It may be possible to obtain a better result by adjusting the

weights in the evaluation function, but the games are taking very long to play, so I will have to consider some sort of automated approach to evaluate performance.

3-22-05

After further consideration, I have come to the conclusion that continuing work on the "evaluation function" would be waste of time. The development of a good function is terribly difficult, and **would be a project in itself**. I do not know enough about chess to even begin the design of a powerful function. Further development of this function does not fit in the goals of this project, and would make very little sense. However, something is clearly required in addition to neural network output. Therefore, I have decided to consider the integration of an existing chess engine—to make use of the built- in evaluation function. Several very strong engines are available for no charge, and take into account a great deal of game parameters which I could never program in the limited time remaining.

The use of a chess engine had another very convincing argument. Without it, judging the performance of the NN system is clearly very difficult, and in all likelihood impossible if any sort of accuracy or objectiveness is desired! Playing against it myself has several problems: my lack of skill, subjective viewpoints, and time required are the greatest.

With a chess engine, I will be able to present the board and obtain a numeric score, without the engine actually deciding on a move. The score will be returned to the chess interface and will be used in determining the move (with the addition of the neural network output—from the previous experiments). The engine is able to play against my program automatically—which is another huge advantage. Now, a non-subjective performance evaluation can take place. Engines allow the ply count (look-ahead length) to be adjusted. Thus, to evaluate the performance of my program, I will compare it to a game with a ply count of zero. After playing several games (in an automated process), I should be able to judge the NN system performance…being better or worse than a ply count of zero (which is the use of an evaluation function only). Although there will be several issues with interfacing, the overall idea is straightforward.

The next task is to select an engine appropriate for the task. I know of two engines, GNU chess and Crafty. I easily find both engines on Google, and download them.

GNUChess comes with a graphical user interface…which makes it unsuitable for use here. I need something with a command line interface. I search on Google briefly for a command line version of the engine only, but do not see one. I decide to search for something else for now, as specific engine used should not matter a great deal.

Crafty seems to have the features I need…it can switch the side to play, accept FEN inputs, and give evaluations. The evaluation function can even be modified (in terms of weights) which is interesting in itself (although probably not important in this project).

Unfortunately, I quickly see that the interface does not produce a move in coordinate notation…it only uses SAN or standard algebraic notation. This output would require that the board be analyzed to determine the start location for each move. Although it can be done fairly easily, it will probably take a good part of a day to set up. It makes more sense to find something which will work "out of the box."

I find a document which specifies the command line interface and instructions for standard chess engines. The document can be found in the sources folder as "Chess Engine Communication Protocol.html." I decide that the actual interfacing will be done in the same manner as was used to communicate between programs in the learning stage. I will use the "script.exe" application and a batch file to start the chess engine and send the proper commands. The command window output will be saved to a text file, which will be done by using "engine.exe > outfile.txt." Once a file is saved, the main interface program will load it and parse the contents to find a score.

3-23-05

I continue to look for a suitable engine. Luckily, I find a very good website: http://www.winboardengines.de/html/winboard-detailpages.html.

This site contains a huge number of engines, although many of them are probably very weak. It takes several attempts to find a suitable engine…and the following are all tried:

Beowulf
Yace
Spike
Bestia
Gothmog
Zappa

All of these engines have something wrong with them which makes them unsuitable…they either do not correctly accept input FEN strings, can not force black to move (they always want to move white), or they do not supply an output in the proper syntax, which will be important for the automated play…

I come across an engine called Gaviota which seems to have everything I need. However, I am somewhat hesitant about using it, as the author himself confesses:

> The evaluation function is slim and is heavily based on the the mobility of the pieces and the pawn structure. King safety is very basic. King likes to go to corner on middlegame and to the center in the endgame.

Thus, I have to question how effective the evaluation function is. Then again, if both sides are given the same function, it should not really matter what it is.

I decide to go with Gaviota for now, and get the engine integrated. If it proves to be horrible for the application, I can always go back ad find something new. Gaviota's command line interface looks like figure 67. The engine seems to comply well with the Chess Engine Communication Protocol.

Figure 67: Example of Gaviota's command line interface

3-24-05

My goal today is to finish integrating Gaviota with my interface. The initial attempt at getting the two to work together is admittedly crude, but will be easy and quickly implemented.

After experimenting with Gaviota, I find that the best way to get a score as an output (an evaluation of a board position which results from black moving) is to enter the FEN string of the new position, with a "w" indicating white to move. **The resulting score will be higher if the position is in white's favor, so we simply multiply by -1**. I disable all castling ability and en-passant square is set to "null." The ply count is ignored. Because the engine will be started fresh each time, it is required to send "sd 1" to prevent the engine from thinking deeply. This saves time, but does not actually impact the score.

"Post" is required to get the engine to write results. In order to enter a FEN string, the command "setboard" is used. An example is:

setboard 7Q/2p2kpp/4Rn2/p2P3P/Pp6/2P5/1P3KP1/8 w - - - 0 0

To generate the fen string, it is required to make a simple function to convert a floating point vector of 64 values into an equivalent FEN notation.

In order to check that the board was received, it can be verified by typing fen at the command line. To obtain a score, "score" is used. A floating point value will print to the screen or output file. When using the scripting application (script.exe), I find that the script shown in figure 68 works correctly.

Figure 68: Script.exe implementation of the FEN string analysis

delay,1
send,*cmd*,sd 1\n\n
send,*cmd*,post\n\n
send,*cmd*,setboard 7Q/2p2kpp/4Rn2/p2P3P/Pp6/2P5/1P3KP1/8 w - - - 0 0\n\n
send,*cmd*,fen\n\n
send,*cmd*,score\n\n
send,*cmd*,quit\n\n

In order to execute this script automatically, I create a simple batch file as shown in figure 69.

Figure 69: Batch file used to execute the script in figure 68

cd "C:\Documents and Settings\Owner\Desktop\NetworksV1"
start script.exe engine_script.scr
gaviota.exe > result.txt
exit

The script is started before Gaviota, because Gaviota opens and quickly closes otherwise. If "wait" is used, the script does not launch. The above solution seems to work. The delay at the top of the script ensures that Gaviota has launched before commands are sent.

Each legal move (Rules_Evals object) will produce a new version of the script shown in figure 68 to reflect the particular move in question. The generated text is sent to a file using the streamwriter class in the constructor of the Rules_Evals class. Once the script is executed, a text file will exist of the command window contents. The text file is then loaded, and parsed to find the score. The score is placed in the "composite_score" property of the object in question. **It is crucial that the streamreader and writer be closed as the same file is overwritten several times during each move!** There are some errors during the code development due to this problem.

In fact, the re-use of the "result.txt" file has caused several problems. I find that the main program is starting execution of a second script before the command windows have closed for the previous one. This is leading to synchronization problems similar to the

problems encountered with the learning process. I decide to fix this by checking for the cmd process and waiting for it to finish before leaving a "while' loop. I also check for the script application, just in case. This technique works, as the program is no longer crashing after I make the changes. The code can be seen in figure 70.

Figure 70: Code to ensure synchronized script execution

```
Process[] id;
id=Process.GetProcessesByName("script.exe");

while (id.Length>0)
{
      id=Process.GetProcessesByName("script.exe");
      //MessageBox.Show("working");
}
id=Process.GetProcessesByName("cmd");
while (id.Length>0)
{
      id=Process.GetProcessesByName("cmd");
      //MessageBox.Show("working");

}
```

At this point, everything appears to be working! The program is successfully returning moves based on scores from the chess engine. Automation must still be added, but this will be done shortly. Now, the big question is how to combine the evaluation score with the NN output. Simply adding will probably not be very suitable, as the evaluation score may fall within a very wide range of magnitudes…0 to 1 is common for early in the game, but values could get up to the 20s or even higher. NN output on the other hand is restricted to -1 to +1. Evaluation scores simply do not have such a defined boundary. The question of how to combine the scores will be based largely on trial and error in all likelihood.

Hopefully, once the white side is implemented to play automatically, it will be possible to obtain the scores over time for several games and obtain averages. When these scores are plotted for a specific combination expression, it should be possible to visualize performance gains or losses and make changes to the function accordingly.

Automation is clearly the next goal. Dr. Malinowski agrees that this direction will probably be the best to pursue, and that competing the engine against itself in the manner described should give an evaluation of relative performance. Several issues do remain however.

First, there is some concern that the engine and NN system will always play the same game. Thus, some way of varying the white starting moves will be required. A few approaches come to mind…a random factor can be introduced to the evaluation function results, or opening books could be used (with random selections). The random aspect should only be required for a few moves to get a good range of sample games.

Dr. Malinowski suggests that specific board positions be used as starting points. These board positions could come from the ChessBase data (randomly chosen). In order to get the best results, it would be required to play the NN system as black and also as white for each position (as one side may have a clear advantage). Although the NN system was trained to play black only, it should be possible to mirror the board and switch the sides. Work on this automation process will continue tomorrow. The code for the newest version of the Rule_Evals module can be found in the journal files folder as Rules_EvalsV4_050324.cs.

3-29-05

The goal for today is to integrate the chess engine with the interface application so that white and black can play automatically. A move randomizer will also be created for the white side (first move only), so that games are varied. A very small random factor will also be added for black's early moves. Without this, the program would be limited to only 20 games. **For now, the approach of playing an entire game is being considered**. Once this is working, adding a feature to start with a given board position will not be difficult. Both the white and black side will move automatically, and should play a whole game when a button is pressed on the interface.

In order to make white move, it is required to have the existing board position sent to the engine, and the search depth must be set to 1. The command to get the move is "go." Essentially, a script will be created in much the same way as is done to get the score for black. A text file will get the command window text, and the text file will be parsed in the same manner (this time, we are interested in the last line—which is the move to be taken).

The code which generates the script and launches it is shown in figure 71.

Figure 71: Code to generate the script to move the white side

```
StreamWriter sw=new StreamWriter("engine_script.scr",false);

sw.WriteLine("delay,1");
sw.WriteLine("send,*cmd*,sd 1\\n\\n");
sw.WriteLine("send,*cmd*,post\\n\\n");
sw.WriteLine("send,*cmd*,setboard "+chessBoard1.getFEN()+" w - - - 0
0\\n\\n");
sw.WriteLine("send,*cmd*,fen\\n\\n");
sw.WriteLine("send,*cmd*,go\\n\\n");
sw.WriteLine("send,*cmd*,quit\\n\\n");
sw.Flush();
sw.Close();
//MessageBox.Show("ok");

Process.Start(@"C:\DocumentandSettings\Owner\Desktop\NetworksV1\start_g
aviota.bat");
```

The code to read the resulting text file and make the move is shown in figure 72. It is actually much simpler than initially thought.

Figure 72: Code to make the white move after parsing the engine output

```
try
{
        string input=sr.ReadLine();
        while (input.IndexOf("move ")<0)
        {
                input=sr.ReadLine();
        }

        the_start=input.Substring(input.IndexOf(" ")+1,2);

        the_end=input.Substring(input.IndexOf(the_start)+2,2);
        //MessageBox.Show(the_start+"    "+the_end);

        chessBoard1.updateBoard(ChessLibrary.Chess.Operation.MOVE,the_sta
rt,the_end);
        chessBoard1.Refresh();
        //output_file.WriteLine("White: "+the_start+"-"+the_end);
}
```

The same technique used previously to make the black side move (updateBoard method) is also used to make the white side move. There are a few problems getting the chessboard control to update itself to recognize the proper side to move (control never switches back to white). However, after further experimentation, it proves *not* to be an issue as the update board method can be used to make any move, and the side in control does not matter. **It can also be used to make illegal moves, but this should not be an issue as a great deal of checking is done between my own checks and the engine**.

At this point, it seems everything is working as expected. When the "Done with turn"—now the "GO" button is pressed, the game begins to play. Black and white take turns—but white has to have the first move made for it. I fix this by adding the move generator code, which simply chooses a move (randomly) from a string array. The function is shown in figure 73.

Figure 73: Code to create a "random" white move.

```
string make_first_move()
{
        int index=rand.Next(19);
        string[]
        moves={"a2","a3","a2","a4","b2","b3","b2","b4","c2","c3","c2","c4
        ","d2","d3","d2","d4","e2","e3","e2","e4","f2","f3","f2","f4","g2
        ","g3","g2","g4","h2","h3","h2","h4","b1","a3","b1","c3","g1","f3
        ","g1","h3"};
        chessBoard1.updateBoard(ChessLibrary.Chess.Operation.MOVE,moves[2
        *index],moves[2*index+1]);
        chessBoard1.Refresh();
        return moves[2*index].ToString()+"-"+moves[2*index+1].ToString();
```

```
}
```

The function is to be called as soon as the button is pressed, but only one time.

Although this works, and a complete game is now played (still need to look at the end game and determination of winner/looser), I notice an inconvenient bug. The entire play routine is actually written in an event handler, and there is no way to stop it (other than using the task manager to stop the process). I talk to Dr. Malinowski, and he suggests that a new thread be started to run the game. I look into this, and find it is actually very straightforward to implement. The button's event handler is modified to the code shown in figure 74.

Figure 74: New button handler
```csharp
private void button1_Click_1(object sender, System.EventArgs e)
{
    button1.Enabled=false;
    playThread = new Thread(new ThreadStart(main_game_process));
    playThread.Start();
}
```

Disabling the button ensures that the thread is not "started twice."

The main_game_process is now a method—exactly the same as the original button event handler.

Testing the new code reveals that everything works as expected. I add a feature to pause the game execution (another button) and also to resume the thread execution. The complete code file for the interface can be found in the journal files folder as InterfaceV4_050329.cs.

3-31-05

An abstract and presentation outline are created which may be seen in the journal files folder as abstract_expo.doc and outline.doc. These are sent to Dr. Anakwa and Dr. Malinowski.

Now that complete games may be played, I am worried that the two sides are not fairly matched. The black side (neural network) is in effect looking "one ply" ahead with the aid of the engine—in the form of an evaluation function output. The question is whether or not the white side has the same advantage. It is crucial that the evaluation scores come from exactly the same search depth, otherwise the performance study will not be accurate.

Although this is a very simple idea, verification proves more difficult. I decide that the best way to find out if the engines are matched is to let the system play a game, and then compare the moves made to the moves made by the engine when presented with the same game. The game is started, and is allowed to play for ten plys. A modification of the code is required for the game to be saved. Code is inserted to write the moves out to a textfile

using the streamWriter class. The output occurs at the same point where the moves are decided and actually made on the board.

Once the game file is saved, it is opened and ~~and the~~ moves are manually entered into the Gaviota engine using the "force" mode and the command "usermove." Search depth is also set to 1. Only the first white move is entered. In theory, the engine should make the same move for the black side—as the neural network output value has been removed from the composite score equation. This is indeed the case. The moves made for the white side should also match, which they do. However, after the first 4 plys are made, the engine does not return the same move! Repeated testing reveals that the engine is NOT playing the game in the manner previously thought.

It is possible that "1 ply" means the engine is looking ahead to the next white move—when I only want it to look at the black position resulting from a possible white move. The other possibility is that the score is not a complete value used in the evaluation of a given board position. In the end, it really does not matter why this is happening—but the methods used in evaluation must match for the black and the white side.

Things are much more complicated as a result of this discovery. It is now required to use the engine to obtain a score in the same way used to obtain the scores for the black side. A discussion with Dr. Malinowski held on Tuesday proves helpful here. We had discussed the proper way to invert the board in order to get my rule logic to work for the white side. Currently, my rule logic will only play correctly as black, as the pawns are not capable of moving from bottom to top. By reversing the colors, and mirroring all pieces over the horizontal line through the middle of the board, it is possible to create a board which is identical in the eyes of the rule logic. The rule logic will still be playing black, but after a move is made the move will be converted by mirroring the start and end position—which means white is effectively moved.

This mirroring technique is vital here. In order to obtain a white move, it is first required to identify all of the legal white moves. This is done by inverting the board that is displayed on the interface **(NOTE: NOT A ROTATION).** Once the board is rotated, a new "Rules_Evals" object is created for every start and end position combination valid for the inverted board. If the Rules_Evals object indicates that the move is legal, an engine script is generated containing the board and the script is executed. This script does exactly the same function as the black side scripts. After parsing of the resulting text file is complete, a floating point score is obtained.

This time, THE LOWEST score is chosen, as the Rules_Evals class has multiplied the score by *-1 in order to make the better scores + (and large) for the black side. After a fair amount of debugging syntax errors and variable name issues, the code is able to compile.

Can it play?

A massive change is noticed within about twenty minutes of play. White's performance has decreased dramatically, and the black side seems at an equal level (it does not currently have the neural network term involved in the calculation of composite scores). Now that the playing field seems to have been leveled, I decide to add the neural network output term again. The composite score is to be calculated by:

Evaluation_Score+.2*Neural_Network_Output

A new game is started. Although both sides are making poor moves on occasion, it seems that black has a huge advantage…in fact, after 29 plys, the black side has taken an obviously overwhelming lead as shown in figure 76. The move history is shown in figure 75. Although it seems to be doing much better, this informal test has raised two very important questions.

1) Several opportunities existed for black to win the game with a single move. No action was taken to end the game. How will I develop the concept of "capturing the enemy king" equally for both sides?
2) White was very severely defeated. So much in fact, that I am now worried that white is not playing with an equal advantage in terms of evaluation score.

Figure 75: History of moves in the test game

```
White Move: h2-h4
Black Move: e7-e6
White Move: f2-f4
Black Move: d8-h4
White Move: g2-g3
Black Move: h4-h1
White Move: f4-f5
Black Move: b7-b6
White Move: f5-f6
Black Move: g8-f6
White Move: g3-g4
Black Move: f6-g4
White Move: a2-a4
Black Move: f8-c5
White Move: c2-c4
Black Move: e8-f8
White Move: a1-a2
Black Move: c5-g1
White Move: c4-c5
Black Move: g1-c5
White Move: a4-a5
Black Move: h1-f1
White Move: e1-f1
Black Move: f8-g8
White Move: a5-a6
Black Move: c8-a6
White Move: b1-a3
Black Move: c5-a3
White Move: f1-g1
```

Figure 76: End position—game was ended at this point

The areas that will need to be looked into in the near future are the above problems, as well as the detection or definition of end games. Dr. Malinowski has suggested that the end game condition be defined by material—and that games be forced to end when the condition is met. This may be the most convenient technique to use for the present situation.

The newest version of the interface code in located in the journal files folder as "InterfaceV5_050331.cs" and the newest Rules_Evals class is named "Rules_Evals_050331.cs."

4-5-05

After looking at the code which is handling the white move, I notice the problem causing the white performance to be so low. The chess engine is performing the evaluation based on the assumption that white is moving up on the board (it has the lower half of the board). When the board was mirrored in order to determine the set of legal moves, it is not being placed back in its original form to be sent to the chess engine. Even though the board DOES have white on the bottom, the engine does not evaluate the position the

same as it does when the original board is applied (I see very high score values in the game output file within the first 2 moves…this should not be the case—values should be 1 or less). I am surprised that this is the case, and decide to fix the apparent problem *by mirroring the board again* prior to creating the script for evaluating a potential board position for the white side.

I also make a minor addition to the Rules_Evals class, which requires an input for "the neural network output" when a new instance is created. This input is normally between 1 and -1. However, when a Rules_Evals object is created for a white move, there is not a neural network output associated with it. Something must still be passed (or an overload could be created). Zero can not be used, as a network could actually return this as valid output for a black move. Therefore, the value "2" will be passed as the input when a white side's Rules_Evals object is instantiated. Now it is possible to generate the script based on this input. Figure 77 shows the choice made in creating the end of the FEN string to be passed to the chess engine. A "b" indicates that black is to make the next move, while a "w" means white will move next.

Figure 77: Deciding whether a FEN should end in "b" or "w" to move

```
if (net_eval>1)
{
      fen=fen+" b - - - 0 0";
}
else
{
      fen=fen+" w - - - 0 0";
}
```

The point of this entire modification is to obtain a board evaluation with an easy to interpret sign. The engine returns a value with reference to the side TO MOVE. **Thus, it is always desirable to make the move with the lowest (or most negative) score as returned from the engine.** The best explanation of this is that the move to be made should be chosen to have the most negative impact on the enemy—which is the SIDE TO MOVE NEXT according to the engine. Without the code in figure 77, it is required to choose the **most negative score for one side**, **while choosing the most positive** when moving for the other (all moves were previously returned with scores in reference to the black side to move). A single function may now be used to pick the best move—and there is less chance of error in misinterpreted signs.

Once this is done, I look at the game's output file (move record) and see that the scores are now more inline with expectations. The reported move does match the best move for white (it is the highest in the list).

I play a few games to see if the performance is improved. The neural network output's weight is set to 0 for the calculation of the composite score. I notice that the two sides are nearly identical in strength after examining 3 games for 40 plies. Although there is no way to know for certain without obtaining a very large data sample, I am convinced both

sides are playing with equal "advantage" contributed by the evaluation function. I will be discussing the further verification of the "control" case shortly.

At this point, it seems to make little sense to worry about the system having an understanding of the "king capture." I will be checking the result of games in post-processing of the saved game files—and looking at material score. The specifics of this will have to be decided on later. When a win is defined in this manner, it is possible to restrict the game length for practical (time) considerations. A loss condition is programmed—which is based on the detection of an empty array of move choices returned for the white side. The loss for the black side is based on the return of the illegal move b8→h1. This is the "filler" value of an initialized Rules_Evals class. If zero moves are legal for a given board position, a vector is returned full of this move (for the black side only). The difference in the detection techniques are of little consequence to the outcome of the game—and are a result of minor coding differences in the creation of white and black move lists. The loss detection is only valid (and needed) if the loss occurs in under 40 plies. Complete code files for the final interface will be provided shortly.

The final task is to implement full automation of the game play. This is very straightforward—and is done completely within the main game process, which is started as a thread when the GO button is pressed. A streamReader object is used to read a line from a file containing a list of FEN strings. The FENnotation property is then set to the string, as shown below:

```
chessBoard1.FENnotation=new_position_from_file.Substring(0,new_position
_from_file.IndexOf(" "));
```

Once the board position is loaded and set, the string is saved to a file holding finished FEN strings. Before a string is loaded, the list of finished games is checked to see the current progress. This allows the interface application to be stopped and started again without losing the progress.

Play in a single game will continue for 40 plies, or until a loss is detected. Once a game ends, the streamWriter object which is saving the game moves will be closed. Then, a new file is created (for the next game), and the next FEN string is read from the positions file. Play continues. The process repeats over and over until the end of the FEN file is reached. There are a few errors initially with getting the next game to start, but I find the reason to be a streamReader and streamWriter have been created at the same time for the finished game list. I fix it by closing the read stream before creating the write stream.

Testing with the game length set to 2 plies indicates that the interface is now working correctly. I change the plycount back to 40, and recompile the FINAL version of the interface. The only change at this point is expected to be the function used to combine the evaluation output with the neural network output. This will be customized for each user involved in the data collection.

4-7-05

Now that the interface has reached a point to be considered complete, the entire project folder is located in the journal files folder as ChessInterface2Final.

I consider allowing the user to change the function used for the composite score calculation, but decide against it. I will hardcode the weights used, and distribute individual copies of the interface application to all users involved in the project. This is to ensure the data obtained is accurate. There is a possibility of weights being improperly entered or changed if the option is left available to the user, which would mean the data obtained would be meaningless.

I will be running the interface on 2 PCs, and will also provide a copy to Dr. Malinowski and Shom. Between the 4 computers, I feel enough data should be attainable to make an accurate observation of the system performance.

In all, 650MB of data and applications are required for running the interface. I decide the best distribution method is by CD. A CD will be created for Dr. Malinowski, which will serve as the "control group" for the testing. The composite score will be set equal to the chess engine's evaluation function. Both the black and the white side should play equally well in this case.

Shom will be given a version of the interface which will compute the composite score by Evaluation_score+10*Neural_Network_Output.

I will be running versions which will compute the composite score based on:
o Evaluation_score+2*Neural_Network_Output
o Evaluation_score+5*Neural_Network_Output

The weights are chosen more or less arbitrarily, and may or may not lead to a good result. Anything that is shown to consistently win over the control group will be considered a success, however the degree of this success will likely change with the weight. If time allows, more weights will be tried once this first round is complete. I decide not to try any weights under 1 as the NN contribution would end up being negligible in comparison to the evaluation function result.

400 FEN strings are included in all distributions (all sets are the same). The strings are obtained from a randomly selected block of games in ChessBase. The positions occurring after 5 plies are saved (this means black is to move).

There is little to do at this point except wait for the game data to be ready. While the data is being processed (games are being played) over the next ten days or so, an application will be created to perform the post processing of the game data. The application will load all saved files, and calculate the running score for the black side and white side. The results will be saved in a comma delimited file for importation into Excel. More discussion of this will follow.

I am now working on completing the first draft of the final presentation, which is being practiced at 6:30 today. The presentation will consist of slides from the proposal presentation, as well as the progress presentation given two weeks ago. A few more are being added (or will be added) that describe the results obtained from the testing which is occurring right now.

The presentation can be found in the journal files folder as FinalPresentation_050407.