Complex Decision Making With Neural Networks: Learning Chess
Project Proposal

Jack Sigan
Bradley University ECE Dept
Dr. Aleksander Malinowski, Advisor
November 28, 2004

Abstract of Project:

Chess has long been one of the most scrutinized, perplexing, and purely logical pursuits of man. Kasparov's defeat at the "hands" of Deep Blue indicates that computers, within a miniscule sixty years of evolution, can outperform the zenith of biological evolution in its crowning achievement of logical decision making.

Yet, not a single machine plays chess in the same manner as man, being capable of little more than brute force attacks on the game. Through experimentation with artificial neural networks with various topologies and learning algorithms, it is hypothesized that learning can be based on "schemas" found in recorded games.

Introduction:

The real question to be asked; can chess really be implemented with ANNs as opposed to the traditional approach of exhaustively searching game trees?

Applied correctly and appropriately designed, artificial neural networks have extraordinary potential for solving problems involving generalization, trend recognition, and pattern matching. Game play, which often involves non-linear strategies or decision making, is a particularly good area to demonstrate the ANN as a way of approximating otherwise inexpressible functions [1]. To date, the promise and lofty expectations of this artificial intelligence approach have yet to be fully realized, demanding further research. Work on complex problem solving, such as that required in classical board games such as chess, has been limited, although many of the available research results [1-4] are tantalizing.

Numerous published studies serve as motivation and a starting point for this research. Chekkapilla and Fogel, in developing an ANN to play checkers, indicated that there is feasibility in teaching ANNs games of some complexity [1,2]. Although chess is clearly a leap forward from checkers, it seems a logical next step in the evolution and development of the ANN, as chess is one of the most widely studied and researched games. A demonstration of strategy in such a game is also recognized as a direct measure of logical decision making ability [1]. Chess as an ANN problem is not a new idea; in fact, ANNs have already been proven to be highly effective in playing chess endgames, although it is ironic that the author also states that chess is too difficult for ANNs to learn in its entirety [3]. The "Distributed Chess Project," when considering the full game of chess, reported approximately 75% accuracy in choosing the "proper" move when confronted with a chess problem external to the training domain. While not fully successful as implemented, the study does appear to indicate that chess schema may be learned by ANNs [4].

The published studies therefore seem to give a mixed opinion of whether a solution is possible in this problem. However, three points must be emphasized here. Technology has improved exponentially since [3] was published in 1994, which allows vastly more complex networks to be implemented. Also, breaking the game down as proposed in this project has not, in the author's knowledge or opinion, been considered or researched before. Finally, the success (and failures) of [4] can only be seen as relevant to the approach used, which is derivation of ANNs through genetic algorithms. A vast training set was apparently not utilized, and external rule control was not applied. Considering the facts, proposed improvements, and possible implications of this kind of research to the field of artificial intelligence, the project is worth pursuing.

System Description:

Although mainly a research endeavor, the end goal of this project is to produce a system based on artificial neural networks (ANNs) which will play chess (as the dark side only) effectively against a human opponent. The dark side is chosen simply because it prevents the system from having to make the first move. To meet the objective, research will be centered on artificial neural network (ANN) topology specifically for the purpose of creating a topology appropriate for complex decision making in a massively dimensional problem. Functionality of the system may be broken into three parts: the learning mode will be an automated process where the ANNs learn how to play from an

extensive external database of games; while the playing and advisory modes of operation accept user interaction, taking move inputs from the player and providing the move(s) chosen by the system. Playing mode returns one move while advisory returns multiple.

The system will operate in the learning mode prior to playing chess or advising. Figure 1 shows the possible modes of operation, along with the interconnection between the modules included in each subsystem. The entire system is composed of software running on multiple PCs simultaneously, and no additional hardware is to be used.

No inputs or outputs are specified for the learning mode, which operates in an essentially automated process. The playing or advisory modes receive move inputs from the player and after evaluation is complete will return the best move(s) chosen by the neural network system. Prior to discussing the playing and training functions, it is necessary to briefly introduce the modules depicted in figure 1 with their corresponding IO functions and give a brief introduction of the two major design paradigms to be considered.

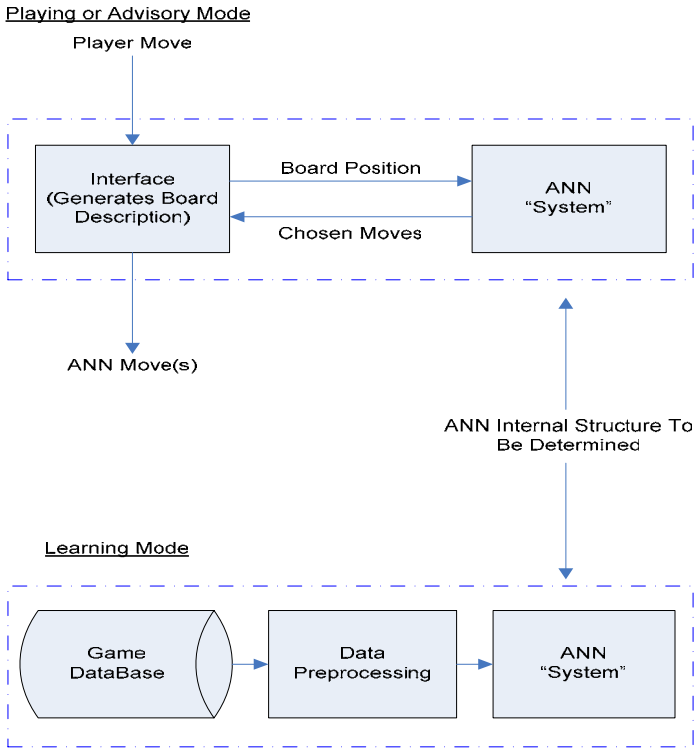

Fig. 1: System block diagram

Input and Output Descriptions:

As shown in figure 1, four main components will make up the system; the interface module, the ANN module, and the preprocessing and database systems. The ANN module is shared by both the learning and the playing/advisory modes. The only difference in the output between regular playing mode and advisory mode is that a list of "good" moves will be returned in advisory mode, while only one will be returned in play mode. Figure 2 displays the connections and IO paths which will comprise the modes of operation. This same information is shown graphically in figure 1. Note that this IO plan

is very high level, and that far more complexity is to be found within the individual modules, which themselves will be broken down into components. Two approaches to the internal design of the ANN module will be presented, which differ in the way chess is broken down for parallel analysis. One paradigm is known as the functional approach, where each piece in chess is considered as a unique unit and has a dedicated network. In the other approach, known as the geographical paradigm, the piece is ignored and instead every possible combination of initial and final positions forming moves is considered a unique unit and receives a dedicated network.

Learning mode:

Module	External Input	External Output	Inter-Modular Input	Inter-Modular Output
Game Database/ Preprocessing	NA	NA	NA	Game records to ANN module
ANN Module	NA	NA	Game records from database/preprocessing	NA

Playing/Advisory mode:

Module	External Input	External Output	Inter-Modular Input	Inter-Modular Output
Interface Module	Player's move	ANN's move(s)	ANN move(s) choice	Board description to ANN
ANN Module	NA	NA	Board description from interface	Move choices to interface

Fig. 2: Input/Output descriptions by module

Interface Module:

The interface module seen in figure 1 (playing or advisory mode) is an application the user interacts with on the PC. It displays a chess board and the piece layout for the game, and the user will make their moves on this board. In advisory mode, a list of returned neural network moves will be displayed in a text box for the user to examine. In playing mode, the computer will make its own moves directly on the board. The interface module must have bidirectional communication with the artificial neural network system (ANN system) to exchange board position data and move choices.

ANN System:

The ANN system is the central focus of the design. It will contain a series of neural networks (as many as 1856 in the geographical approach), operating in parallel, which are trained to play specific moves in the game, or as specific pieces in the game depending on the design paradigm in question. Before use in the playing/advisory modes, the ANN system must be trained in the learning mode, where it starts as a randomly connected, randomly weighted network. The training takes place in an automated process utilizing millions of data samples representing “good” moves in chess. Once trained, the ANN system is used by the interface module to play the game. A board position may be passed to the system, and a list of suggested moves will be returned. The interface decides whether to make one move or to display a list of suggestions based on whether it is set to playing mode or advisory mode.

Game Database:

The game database is a standalone application called ChessBase 9.0. This application allows a massive database of several million saved chess games to be sorted, filtered and searched by move. ChessBase 9.0 is used exclusively in the learning mode, at the start of the preprocessing stage. All game data must be preprocessed in order to convert it from the database format to a format usable for neural network training. Games from ChessBase 9.0 are extracted by searching for specific moves or movement by piece, which defines the first step in the preprocessing module to be described shortly. Many individual sets of games are made, which will then be passed to the remaining preprocessing stages.

Data Preprocessing:

Data preprocessing begins in the game database described above, and continues with a series of applications designed to convert the chosen game data into neural network training vectors. The data processing stage converts PGN files (portable game notation—a popular algebraic game recording standard) which are extracted from the database, to EPD strings (extended position description). EPD files contain a series of strings representing the full board position at each move recorded in the PGN file. Figure 3 shows a sample of PGN format, and figure 4 shows a sample of EPD format. At this time a detailed understanding of file content is not required.

```
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
```

Fig. 3: Sample of the PGN file standard

```
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;
```

Fig. 4: Sample of the EPD file standard

Of course, neural networks require a numeric input vector for training. It is therefore required to convert the EPD stings into floating point input vectors by converting the EPD characters to floating point values. A sample geographical training vector is shown in figure 5. The output pattern is 1 or -1, corresponding to whether or not to make the move specific to the training data set and the neural network. The hope is that after learning enough patterns, the neural network will be able to effectively make move decisions for data samples which are not contained in the original training set. Training vectors for the functional approach will have multiple output patterns, which will be clarified shortly.

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

Fig. 5: Sample floating point training sample

It may be possible that training with +1 and -1 only is not the best approach to training after some further consideration. 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, the concept of "move strength" will not be pursued further. However, the possibility of adding additional network inputs at a future time remains open so long as they are clearly visible on the board or in the game. Perhaps piece proximity data, last move made, or some other data could be valuable to include in the training vectors and include in the eventual decision making process. It may also prove more effective to train with values of +0.9 and -0.9 in order to increase learning speed. Now that the main modules have been adequately introduced, it is now possible to describe the full functionality of specific modes of operation.

Playing/Advisory System:

The interface module and the ANN system are used in the playing/advisory system. Figure 6 describes the operation of this entire system (or functional mode). In figure 6, the ANN board processing stage is expanded on the far right to give more insight into the ANN system. It is possible to treat the ANN processing as a single function as it has a clearly defined input and output, and its inner workings are of no consequence to the interface module so long as data is passed back and forth. The expanded ANN processing function can be better understood by figure 7, which shows the block diagram of the geographical ANN system design. Here, "adaptive resonance" is simply a specialized training session held after the initial training to "fine tune" networks so they make fewer illegal moves over time.

The true form of adaptive resonance would make changes to the network during training, while in this case, although the data is saved within the playing/advisory mode, it is required to go back to the training mode to complete the "adaptive resonance." Thus, while not adhering to the true definition of adaptive resonance, the end result is the same; adjustment of weights to differentiate between similar inputs based on observation of a logical error.

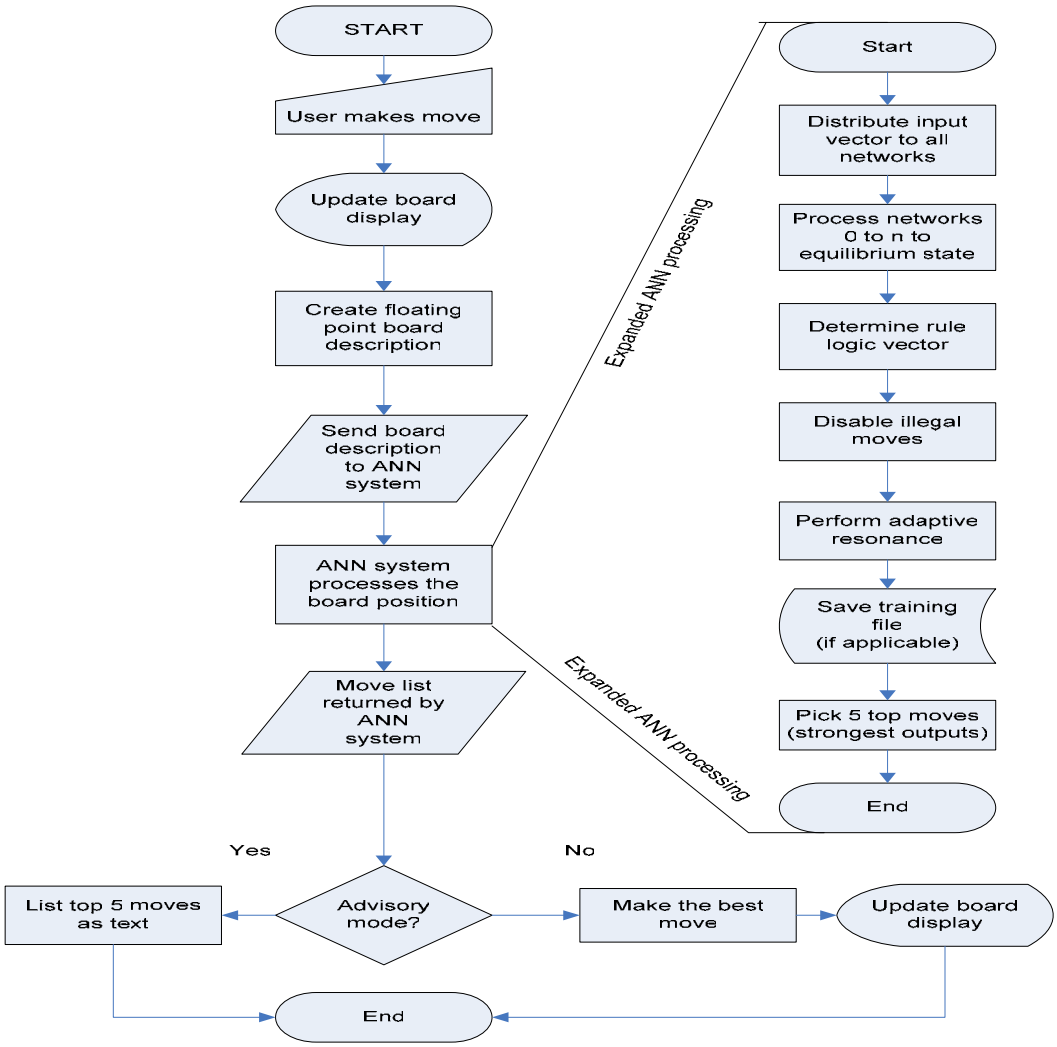

Fig. 6: Playing/advisory system flowchart

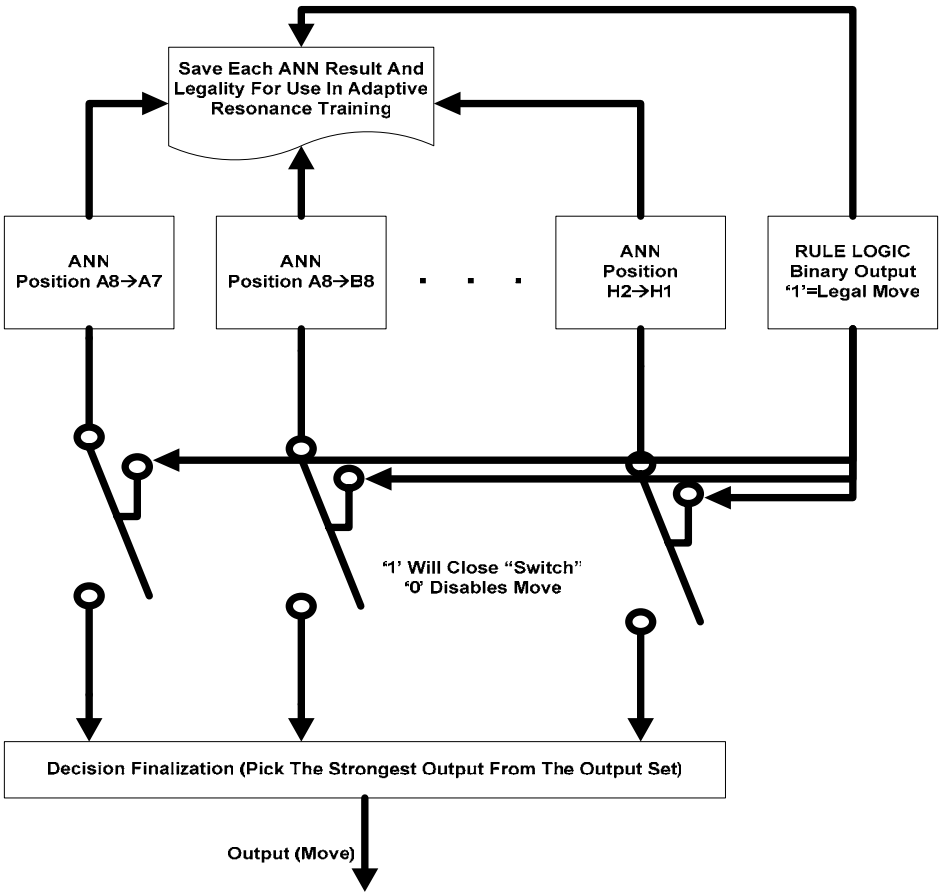

Fig. 7: Move based ANN system design (geographical design paradigm)

Learning System:

The learning system is comprised of the ChessBase 9.0 database, the preprocessing software, and the ANN system. It is possible to divide learning into two distinct functions: data preprocessing and training.

The data preprocessing function is described in figure 8. The database specific operations and selections can be considered a unique function, and an expanded view is provided at the bottom of figure 8. It is vital that the final output of the preprocessing stage is a set of training vectors specifically created for each possible legal move in chess, following the data format displayed in figure 5. A great deal of research and effort has been devoted to the preprocessing stage to develop it to this point, as the functionality required does not exist in any known software. It is required to use numerous applications to convert between formats, the names of which are displayed in the appropriate functional locations in figure 8 below. In some circumstances, another application may be required to break large PGN files into small files prior to running EPD_Position.exe. However this is not shown in figure 8 as it is in place only to improve speed.

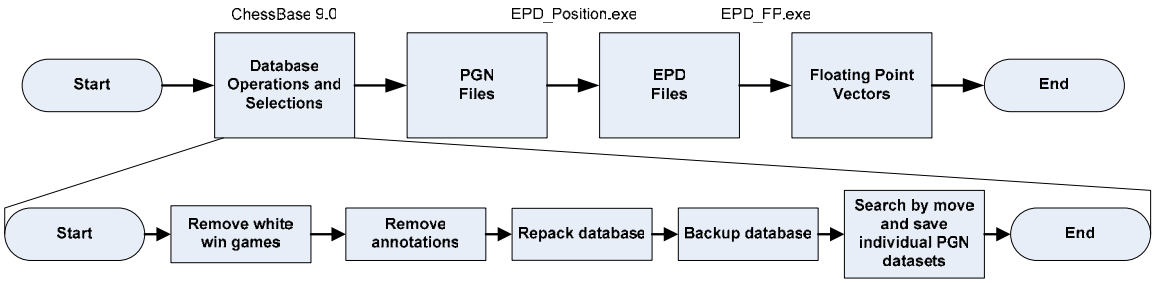

Fig. 8: Data preprocessing subsystem flowchart

Training takes place after the data preprocessing is complete. The end result of preprocessing is a set of "pattern files" which are loaded by the Stuttgart Neural Network Simulator (SNNS) to perform the training. Each network has its own pattern set, and training takes place one network at a time. Each training pattern is once again in the format described in figure 5. Training is performed entirely by SNNS, so the mathematical details of learning will not be described in this document. It is possible to instead treat the actual learning as a self contained process. Figure 9 describes the training process.

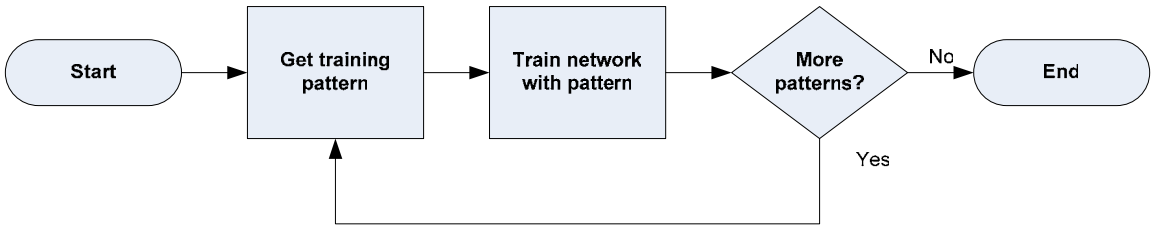

Fig. 9: Training process flowchart, using Stuttgart Neural Network Simulator

Training will actually be repeated numerous times for each network. Each time the entire pattern set is trained is known as a cycle. Approximately 100 cycles will be processed for each network, which is subject to change based on observed training effectiveness.

Structure of the ANN Modules:

Now that all components, functions and design paradigms have been introduced, the internal structure of the ANN module will be discussed in greater detail. Figure 1 depicts the ANN module in both the learning and playing/advisory mode, and it must be emphasized that the design of this component is the primary focus of the project's research. Presently, two major design paradigms are being examined.

The first will involve a geographical breakdown of all possible moves in the game based on the starting position *i* and the final position *f*. An entire game *g* of *n* moves may be expressed as a set of board positions b_i where *i* is the move number from 0 to *n*. Of course, each $b_i \epsilon g$ has a set of possible legal moves based on the piece *p* located at all *i* positions held by the game participant's side. Therefore, it is possible to define the set of

legal moves as $m_{if}=f(b_i)$ since complete knowledge of *p, i,* and *f* may be obtained from any b_i in addition to the rules of chess. For now it is required to consider the castling moves to be special cases, as they involve more than 1 piece. If considering a whole game of *n* moves, then all legal moves made throughout may be expressed as $L = \sum_{i=0}^{n} f(b_i)$ All legal moves in chess may therefore be found as $M = \sum_{i=0}^{\infty}(L_i)$. It is not difficult to determine m_{if} through simple logic for any $b_i \in g$. *M* is finite, obviously having a value less than 64x63 moves. This idea is fundamental to justifying the design in figure 7. In this design, it is required to create an individual ANN structure for all moves *t*, $t \in M$. Thus, each ANN will be trained to make a 'yes' or 'no' decision for its own move *t* based only on b_i. It is possible that some networks may say 'yes' for making a move, even if the move is illegal based on the rules of chess for the given b_i ,even though $m_{if} \in M$! For example, the move for the left black rook may say 'yes' to move from square D4 to D6 even though square D5 is held by an enemy. D4 to D5 may also report a 'yes' decision, so how will an action be chosen? This is where knowledge of m_{if} as defined above for the specific board position is applied. It is possible to deactivate the ANNs which want to make illegal moves. Once this is done, the ANN with the highest output level (from a hyperbolic tangent function, range +/- 1) is chosen as the move to make.

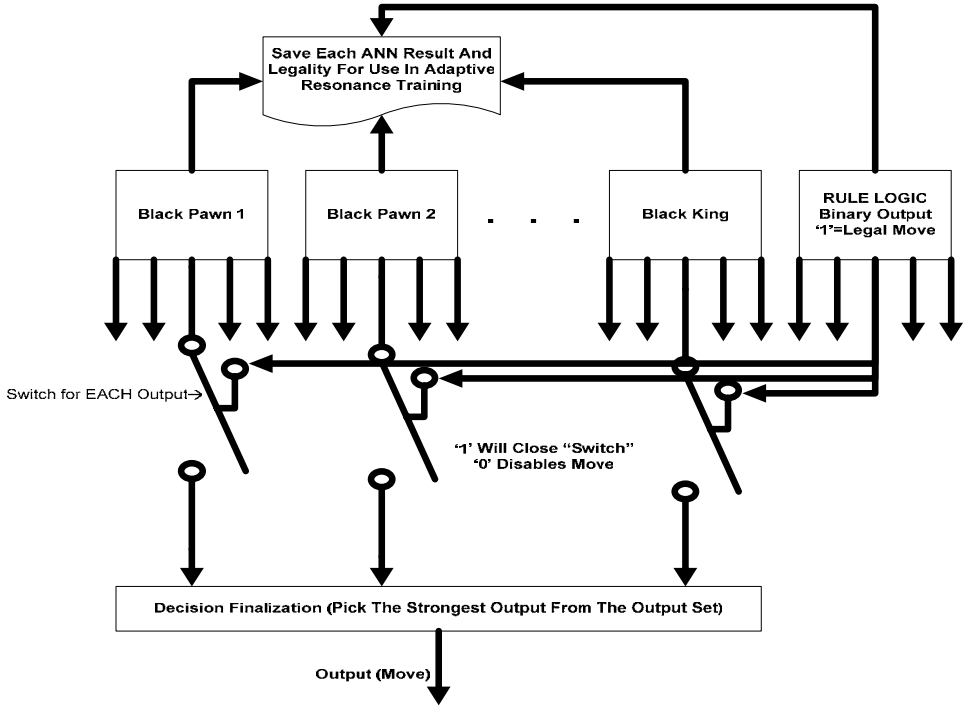

Fig. 10: Functionally Derived ANN System Design

Over time, it should in theory be possible to modify the ANNs so they do not make these illegal moves anymore by using an implementation of adaptive resonance. In short, the input b_i will be saved in a file along with a 'no' output for any illegal moves. A special training session will then be held to improve the network.

The second approach initially proposed is to break the game of chess apart by piece, thereby allowing a more "functional" approach to the same problem. It is required to define 16 networks which represent all pieces for the dark side. Each network will have m outputs, which are defined by $m = \sum_{i=0}^{\infty}(l_i)$ where $l = \sum_{i=0}^{n} f(p,b_i)$. Defining l is only slightly different from how L was defined above, as it must have emphasis placed on p (the piece) in addition to b_i. Once again n is the number of turns in a single game. There will still be the same number of outputs in total (M). Outputs will still be deactivated in the same fashion as described above. Adaptive resonance will still be applied to improve the network decisions over time. Figure 10 shows a block diagram of the functionally derived approach.

In general, both design approaches may be viewed as parallel approaches to complex problem solving. Many benefits are achieved in a parallel approach, including shortened training time and simplified internal network design. Although it is not known in advance how large the networks must be for either approach, it seems that the networks for the geographical approach will be significantly smaller (in terms of node count) than those in the functional design, simply because they have only one output, thus the simulated function is assumed to be less complex, thereby requiring fewer nodes to achieve accurate approximation results. It is expected that all networks will be the same size for the geographic approach, although it may also seem valid that the moves involving edge positions as initial positions will not need as many nodes as they simply do not have as many legal moves to make, while moves in the center could require a greater number of nodes. Unfortunately, theory does not exist to accurately predict ANN size for problems such as this, so it will be left largely to trial and error. Functional networks clearly will have varying sizes, simply because queen movement, for example, is more complex than that of the pawn.

Another benefit of the parallel approach is that learning is anticipated to be less destructive than it would be for a single network having to process all moves. In such a situation, involving a highly dimensional problem and dataset, adjusting the weights and thresholds to make one output converge could cause divergence of the other outputs. Not only would learning take much longer, but it would seem that accurate results may be far more difficult to come by.

Either approach will be trained using the same dataset, which is a database of several million recorded games of varying skill levels. An interesting characteristic is that the level of play the ANN is capable of may be related to the skill level of the games used in training. If so, then it would be possible to pre-specify the quality of performance when training the network! Although the initial goal is to simply train the system to play without regard to skill level, this could be a very interesting area to look into if time and success permits.

Progress and Current Direction:

A major focus over the summer of 2004 was to develop a generic and versatile ANN framework, including all processing, learning, and management functionality. Some time has also been spent determining the best way to store training records, including the data representation, as well as normalization and standardization

considerations. Although work on the ANN platform is estimated to be roughly 80% complete at this point, the choice was made recently to discontinue work on the ANN framework and instead utilize the existing Stuttgart Neural Network Simulator (SNNS) with Java interface [5]. Work is now focusing on network topology design, network size considerations, and integrating the rule logic with the ANN components.

It is required to have some form of data processing in addition to the ChessBase 9.0 software being utilized for the training data. The processing function will take the games stored in PGN files (portable game notation, a common algebraic chess notation standard) and convert them to extended position description (EPD) format. The result for one game would be a series of strings with one string for each board position in the game. These strings are then to be parsed by another application which will convert them into a series of 64 signed floating point values corresponding to the traditional weights given to chess pieces. The value of the move to be made will be concatenated with the resulting vector to produce a training record. Once a set of training records is generated, they will be presented randomly to the ANNs for training.

The networks themselves will be generated to work with SNNS, as will the training records. Multiple PCs will process the training files in the learning stage, and human supervision will not be required. It is expected that learning will take hours to a couple of days for the entire system. Rule logic is not considered in the initial training process.

Currently, the approach being most seriously considered is to create all networks the same size, and large enough to deal with the largest of the datasets (which is still based on estimation). Then, all networks will be initialized to have perhaps 10% connectivity (they are currently considered 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 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, it is expected 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 it is not desirable to simply stop “halfway through” the data, for example. Because the overly trained network only has 10% connectivity, it is possible to 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.

The networks themselves are currently being considered with hyperbolic tangent activation functions, using classical back propagation learning. It may become clear in the near future that another learning style is more appropriate (perhaps back propagation with momentum, etc), but for now as many simplifications as possible are being sought, so learning will be kept simple for the time being. Positive elements on the input vectors will be scaled to the dark side (ANN) while negative elements will be scaled to the light (human) player. A zero means no piece is present on the square in question. The outputs of the ANN (tanh) are to be considered ‘yes’ if they are positive, and ‘no’ if they are

negative. More positive moves (approaching +1.0) are the ones most likely to be made, while the most negative (-1.0) are illegal or very poor can never be made. As research and design continues, it will doubtlessly become apparent that numerous additions and modifications to the preliminary concepts will be required.

Currently, research is also being undertaken in the area of radial basis function networks (RBFNs) as a possible alternative approach to the feed forward networks currently being pursued. 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 approximation (or 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. RBFNs differ greatly from the typical feed forward approach to neural networks because each node performs a somewhat unique "input summing" operation. Instead of merely adding the inputs multiplied by their respective weights, the RBFN nodes determine the input vector's "distance" from the weight vector via the dot product. If two vectors have "zero" distance from one another, the output is maximum as the Gaussian distribution function has the maximum value at zero. Figure 11 shows the RBFN node and the Gaussian distribution activation function.

If the decision is made to utilize RBFNs instead of the current approach, numerous changes to network topology will be required. Instead of having numerous feed forward layers, the RBFN will have only an input layer, a single layer of RBFN nodes, and a linear summing layer. Connectivity in this case would likely be full rather than partial for the simple reason that hidden nodes will never have the opportunity to "communicate" as they do in the feed forward design as each node is connected straight to the output layer and never to another hidden node.

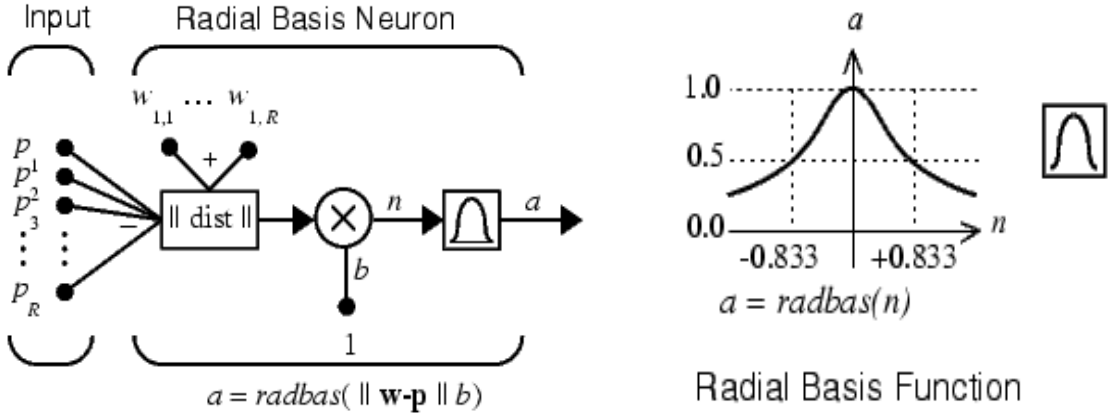

Fig. 11: RBFN node and activation function

A separate application must also be created for the interface to be used in the playing mode. The application should present the user with a graphical view of the board, and be able to read the SNNS files and process the network after a new input vector is generated when the user makes a move. The resulting move must then be shown on the board.

Appendixes:

Schedule and timeline:

The timeline for this project is shown in figure 12. Work began in May 2004 and is scheduled to be completed in the beginning of May, 2005. Thus far, work is on schedule although the data extraction is taking longer than anticipated. There is some danger that it could be delayed depending on the time needed to extract the center moves, which do take a great deal longer than the edge moves. However, this step is the slowest and most time consuming task in the project and after it is completed everything will be back on schedule.

Date	Goals and progress
May-04	Decide overall purpose of the project
Jun-04	Work on neural network framework
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 software
21-Oct-04	Network generator program is created and data processing defined further
28-Oct-04	Order ChessBase 9.0 and evaluate training speeds to estimate needed CPU time
4-Nov-04	Argonne presentation and ChessBase 9.0 arrives. Begin data processing (Extraction).
11-Nov-04	Extract Data. Begin investigating possible network sizes and connectionisms.
18-Nov-04	Extract Data. Begin investigating radial basis function networks and "error" determination.
25-Nov-04	Extract Data, work on proposal
2-Dec-04	Extract Data, Proposal presentation
9-Dec-04	Extract Data and begin to look at feed forward and radial basis comparison
16-Dec-04	Extract Data and continue 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 PGN). Create and initialize all networks. Journal paper?
13-Jan-04	Design a process for training and test this on 4 or 5 machines. Journal Paper?
20-Jan-04	Train on maximum number of PCs, evaluate performance
27-Jan-04	Make changes to topology, data format, etc. as needed.
3-Feb-04	Train on maximum number of PCs, evaluate performance
10-Feb-04	Make changes to topology, data format, etc. as needed.
17-Feb-04	Train on maximum number of PCs, evaluate performance
24-Feb-04	Make changes to topology, data format, etc. as needed.
3-Mar-04	Train on maximum number of PCs, evaluate performance
10-Mar-04	Make changes to topology, data format, etc. as needed.
17-Mar-04	Train on maximum number of PCs, evaluate performance
24-Mar-04	Make changes to topology, data format, etc. as needed.
31-Mar-04	Integrate all remaining modules (final interface). Test system against human players.
7-Apr-04	Continue testing system. Evaluate rating if possible.
14-Apr-04	Begin preparing for final presentations and expo + finish loose ends
21-Apr-04	Begin preparing for final presentations and expo + finish loose ends
28-Apr-04	Begin preparing for final presentations and expo + finish loose ends
5-May-04	Project Complete

Fig. 12: Proposed schedule and timeline

Parts and prices:

ChessBase 9.0 Database: $389.00 (ordered and in use)

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 follow shortly.)

References:

[1] K. Chellapilla and D.B. Fogel. “Evolution, Neural Networks, Games and Intelligence.” *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1471-1496, Sept. 1999.

[2] K. Chellapilla and D.B. Fogel. "Anaconda Defeats Hoyle 6-0: A Case Study Competing an Evolved Checkers Program Against Commercially Available Software,” in *Proceedings of the 2000 Congress on Evolutionary Computation*, 2000, vol. 2, pp. 857-863.

[3] C. Posthoff, S. Schawelski and M. Schlosser. “Neural Network Learning In a Chess Endgame,” in *IEEE World Congress on Computational Intelligence*, 1994, vol. 5, pp. 3420-3425.

[4] R. Seliger. “The Distributed Chess Project.” Internet: http://neural-chess.netfirms.com/HTML/project.html, 2003 [Aug. 26, 2004].

[5] University of Tübingen. “Stuttgart Neural Network Simulator.” Internet: http://www-ra.informatik.uni-tuebingen.de/software/JavaNNS/welcome_e.html, 2003 [Aug 30, 2004].

[6] M. Chester. *Neural Networks*. Englewood Cliffs, NJ: PTR Prentice Hall, 1993, pp. 71-81.