Evolving intelligent game-playing agents

NELIS FRANKEN AND ANDRIES P ENGELBRECHT
Department of Computer Science
University of Pretoria

Traditional game playing programs have relied on advanced search algorithms and hand-tuned evaluation functions to play 'intelligently'. A historical overview of these techniques is provided, followed by a revealing look at recent developments in co-evolutionary strategies to facilitate game learning. The use of particle swarms in conjunction with neural networks to learn how to play tic-tac-toe is experimentally compared to current game learning research. The use of a new particle swarm neighbourhood structure and innovative board state representation show promising results that warrant further investigation to its application in more complex games like checkers.

Categories and Subject Descriptors: I.2.6 [**Computing Methodologies**]: Artificial Intelligence - *Learning*;
General Terms: Algorithms, Design, Experimentation
Additional Key Words and Phrases: Artificial intelligence, game learning, co-evolution, particle swarm optimisation.

1. HISTORICAL OVERVIEW

Since the dawn of Artificial Intelligence (A.I.) techniques, researchers have aimed to create algorithms and program code that seemingly play games *intelligently*. One of the pioneers in this field was Arthur Samuel, who envisioned an intelligent program competing against world-class human opponents in a game of checkers [Samuel 1959]. Game learning has arguably been one of the biggest success stories of A.I. research to date [Schaeffer 2001], receiving widespread attention from leading scientists around the world.

Researchers have historically divided the different types of games into two main categories, namely zero-sum and non-zero-sum games. The former is characterized by the fact that all the game information is available at all times from the start to the finish of a particular game. For example, all the pieces that are still in play during a chess game are visible at all times. This category is also often referred to as perfect information games, for this specific reason. Other examples include checkers and backgammon.

Non-zero-sum games have a hidden component. Formally stated, it means that not all the information about the game is available to all players at any given moment during the progression of the game. Examples of these types of games include poker and bridge.

Advances in search algorithms and tremendous advances in hardware capabilities have boosted the power of these game-playing machines. One of the biggest advances culminated in the two public tournament duals between the IBM supercomputer 'Deep Blue' and world chess champion Garry Kasparov in 1996 and 1997. 'Deep Blue' became the first computerized chess player to beat a reigning chess grandmaster in a regulation match [Schaeffer 2001].

After the success of 'Deep Blue' many researchers have stepped away from chess as their favourite test bed and have resumed research into other games [De Coste, 1997; Chellapilla et al. 1999; Fogel 2001]. The Holy Grail for A.I. researchers in the gaming arena still remains the ancient Chinese game of Wei Chi – commonly known in the West by its Japanese name: *Go*. It has been estimated that *Go* has a search space (all the possible combinations of legal moves in a standard game) upwards of 10^{170} moves. Comparatively, chess has approximately 10^{40} legal moves [Churchill 2001]. This fact alone puts *Go* in a class of its own, and for the moment remains unsolvable using brute-force and traditional search-based A.I. techniques.

The phrase 'solving a game' refers to a computer discovering how to play a perfect game from start to finish, always making the best move in any given situation – thereby never losing a match. Solved games include Nine Men's Morris, Connect-4, Qubic, and Go Moku [Schaeffer 2001].

This paper investigates the use of particle swarm optimisation and co-evolutionary techniques to train superior tic-tac-toe game playing agents. Section 2 takes a more in-depth look at some of the more popular methods and recent

Author Addresses:
CJ. Franken, Department of Computer Science, University of Pretoria, Pretoria, 0002, South Africa; nfranken@cs.up.ac.za.
AP. Engelbrecht, Department of Computer Science, University of Pretoria, Pretoria, 0002, South Africa; engel@driesie.cs.up.ac.za.

developments in game learning research. Co-evolution is introduced in section 3, followed by an overview of standard particle swarm optimisation techniques in section 4. Section 5 ties everything together by looking at the training algorithm and the configuration of the neural networks and particle swarm. Section 6 provides some interesting experimental results for training agents to play tic-tac-toe. Finally, section 7 briefly looks at future work resulting from this research.

2. GAME-TREES, CHECKERS AND EVOLUTION

Traditional A.I. methods that were developed to increase the skill of computers to play zero-sum games have mainly focused on advanced search techniques. Typically, a human playing a game of checkers will try to 'think ahead' a couple of moves before making his final move selection, constantly trying to set up a series of moves leading to a winning solution. Bluntly stated, the further ahead a player can 'see' into the progression of a game, the easier it becomes for the player to steer the game into an advantageous situation, ultimately (but not always) resulting in a win.

This technique has also been applied to A.I.-based computer opponents. The computer will try to construct every possible move it can make, given the current board state. It will then alternate to predict all the possible moves its opponent can make for each of its original predictions. The information is stored in a tree data structure commonly known as a MIN-MAX game tree [Nilsson 1998]. This series of operations will continue until some stopping-criterion is met, usually defined as a depth limit to the tree (limiting the number of turns the program is able to 'see' ahead) or a time-limit (regulation chess matches usually have a 3 minute time cap per move imposed on each player).

After the tree is constructed, each generated board state has to be evaluated in order to determine whether a certain move is 'better' than the next. Depending on the evaluation for the different board states, a path can be constructed from the initial (current) board state to the most advantageous board state. Two of the major obstacles in constructing this path involve the technique used to search through this tree, and the methods employed to evaluate each board state.

Various optimisations have been developed to search through a game tree - the most well known being the popular alpha-beta search [Nilsson 1998] and NegaScout [Reineveld 1989]. The evaluation of the board state has always been highly dependent on the specific game implementation. The team of dedicated researchers working on 'Deep Blue' implemented evaluation functions in hardware to speed up the process, while working with chess Grand Masters to determine positive features of a game board.

The reigning software checkers world champion, 'Chinook' [Schaeffer et al. 2000], also relies heavily on databases of opening-moves and end games (as did 'Deep Blue'), consisting of thousands of annotated high-skill checkers matches between grand masters over decades of checkers history. In effect, 'Chinook' was given vast amounts of expert checkers knowledge and programmed to search through it as fast as possible.

Some people may well contest the validity of labelling these techniques as being *intelligent*. Samuel himself used a manually-tuned evaluation function for his checkers program [Fogel 2001], but stated that he believed the true study of intelligence would come from programs that learn to play the game without any human interference or guidance as to what constituted a 'good move' or 'winning strategy'.

Recently, more and more researchers are showing interest in using 'soft-A.I.' approaches to augment the traditional 'hard-A.I.' solutions. Kendall and Whitwell [2001] use evolutionary algorithms to fine-tune a chess evaluation function. Fogel and Chellapilla take the whole process one step closer to Samuel's vision by opting not to use a handcrafted evaluation function. Instead, their checkers program codenamed 'Anaconda' (or 'Blondie24', depending on its use) rely on the 'black box' approach of neural networks in combination with evolutionary training strategies – to great academic and commercial success [Chellapilla et al 1999, 2000; Fogel 2001].

3. CO-EVOLUTION IN A NUTSHELL

Neural networks are better known for their use in supervised learning [Engelbrecht 2002], where the outcome of a given input fed through the network is matched against an expected output. The difference between the *actual* output and the *expected* output translates to an error-value, which is used in back-propagation neural networks to alter the weights that connect the output layer to the hidden layer, and the hidden layer to the input layer. This process usually continues iteratively until a low-enough error-value is reached.

Using neural networks in their originally intended form for game learning has one major disadvantage: there is no expected output to compare against. If a comparison could be made, then it meant that the computer knew exactly what the best move in the current situation should be, which would only happen in 'solved' games – defying the purpose of learning any playing strategy for that particular game.

To counteract this shortcoming, a technique known as co-evolution is employed to train the neural networks (game playing agents). Co-evolution can easily be illustrated by a competitive relationship in nature [Engelbrecht 2002], where a plant has to adapt (evolve) in order to survive the continued attacks by native ant species. The plant excretes fluids to harden its defences, and the ant evolves stronger jaws to penetrate these defences. This quid-pro-quo relationship continues indefinitely, and the species co-evolve into stronger/better adversaries.

Applying co-evolution in a game learning context requires that the agents compete against each other in some tournament scheme, tally their scores, compare their results against the overall 'winner' for that generation and subsequently adjust their playing style to increase performance in future competitions. The agents drive their own learning (evolution) through competition with other agents - hence the term, co-evolution.

4. ADDING INTELLIGENT SWARMS TO THE EQUATION

For the previous discussion it was simple enough to just state that the agents 'adjust their playing style'. The actual process involves adjusting neural network weights to approximate the evaluation function of the specific game more accurately. Adjusting these weights to reach an optimal solution can be achieved through various techniques. Fogel and Chellapilla [1999] used evolutionary programs (EP) to adjust the weights, applying standard evolutionary constructs such as selection, reproduction and mutation to improve and diversify their individual agents.

A new optimising technique has been developed in recent years - inspired by the flocking behaviour of birds. Kennedy and Eberhart [1995] first proposed the Particle Swarm Optimisation (PSO) algorithm in 1995. Since its inception it has revolutionised various engineering fields and has been applied in computationally dominant research areas to optimise highly dimensional mathematical functions (among other things). In almost all cases PSO compares well with EP implementations, often significantly surpassing EP performance. PSO has already proved successful in training neural networks [Van den Bergh et al. 2001] and has recently been applied to the game learning research domain [Messerschmidt et al. 2002].

In short, PSO involves 'flying' a population of particles (candidate solutions to the optimisation problem) through an n-dimensional search space. Each particle has a unique velocity, and keeps track of its best personal position in the search thus far. The particles share information with each other regarding more optimal solutions that have been found globally, and each particle's velocity is updated to 'pull' it towards these better solutions.

Various structures have been developed to facilitate this information sharing process, resulting in significant performance differences depending on the optimisation problem [Kennedy et al. 2002]. In the global best (gbest) implementation of the algorithm, the single best performing particle in the whole population influences the velocity updates. All the particles are 'pulled' towards this single solution. While flying towards the solution a particle might come across an even better optimum, which will cause the rest of the population to fly towards this new position. The gbest implementation is highly susceptible to local optima.

Another approach is known as the local best (lbest) implementation. In lbest, the population is subdivided into neighbourhoods of a predefined size. For each neighbourhood, the local best particle is computed and only this particle influences the velocity updates for its neighbouring particles. Gbest can be seen as lbest with the neighbourhood size set equal to the population size. Lbest is less susceptible to local optima when compared to gbest, but is generally slower.

One of the latest developments in PSO information sharing is known as the Von Neumann neighbourhood structure [Kennedy et al. 2002]. It is similar to lbest in its implementation, with the only difference being how the neighbourhood is constructed. For lbest, a one-dimensional 'sliding window' equal to the neighbourhood size moves across the circular list of particle instances. For example, assuming an inclusive neighbourhood size of 5 particles, a single particle will consider the 2 particles towards its left as well as the 2 particles on its right as its neighbours. The Von Neumann implementation converts the one-dimensional lattice into a two-dimensional lattice. A single particle is now capable of looking at its left and right neighbours (as in lbest), but also compare itself with the neighbouring particles located above and below it.

It is important to mention that quite a lot of parameters are involved in the inner dynamics of the velocity updates, that can with minor tweaking drastically improve the performance of the swarm. These parameters include personal (c1) and social (c2) acceleration constants, an inertia weight and a limit to the maximum velocity (vMax) for each particle.

In the context of training game playing agents, each particle will consist of a large vector representing all the weight values in a neural network. The traditional back-propagation method used to update neural network weights is replaced by 'flying' these particles through the search space. This results in updates occurring in relation to better performing agents' neural networks (particles) – causing them to evaluate the state of the game board more accurately, and in turn resulting in better playing agents.

5. PUTTING IT ALL TOGETHER

A collection of experiments have been conducted to test the performance of various PSO algorithms to train neural networks for game board evaluation, as well as the resultant performance when compared to a player making random moves. The well-known game of tic-tac-toe (noughts and crosses) is used as a test case, because of its simple game rules and relatively small search space.

Tic-tac-toe is played on an initially empty 3x3 grid. It is a two-player game, where players are represented by crosses ('x') and circles ('o') respectively. The players take turns to place their individual symbol in one of the open

spaces of the grid. As soon as three identical symbols line up horizontally, vertically or diagonally, the corresponding player has won the game. If there are no open spaces on the grid, and no symbols have lined up, the game is drawn.

5.1 Simulation algorithm

The following algorithm briefly outlines the steps involved in each experiment:

1. Instantiate population of agents.
2. Repeat for 500 epochs:
   A. Add each agent's personal best neural network configuration to the population.

   B. For each individual in the population:
      i. Randomly select 5 opponents and play a game of tic-tac-toe against each, always starting as 'player one'.
         a. Generate all single valid moves from the current board state.
         b. Evaluate each board state using the neural network.
         c. Select the highest scoring board state, and move appropriately.
         d. Test if game has been won or drawn.
         e. Return to 'a', switching players, until game over.

      ii. Assign +1 point for a win, -2 for a loss, and 0 for a draw after every game.

   C. Compute best performing particle according to PSO algorithm in use.

   D. For each agent (excluding personal best) in the population do:
      i. Compare performance against personal best configuration's performance.
      ii. Compare performance against neighbourhood's best particle.
      iii. Update velocity for each particle according to PSO algorithm.
      iv. Update weights according to PSO algorithm.

3. Determine single best performing agent in whole population.
4. Best agent plays 10000 games against a random moving player as 'player one'.
5. Best agent plays 10000 games against a random moving player as 'player two'.
6. Return to step 1 until 30 simulations have been completed.
7. Compute confidence interval over the 30 completed simulations.
8. Compute performance value over the 30 completed simulations.

5.2 Measuring performance

Messerschmidt et al. [2002] computed the probabilities for winning a tic-tac-toe game using a game tree to be 58.8% playing as player one, and 28.8% playing as player two. His experiments have also shown that when two randomly moving agents play 10000 games against each other, the ratio of games won as player one and player two correspond to these game tree probabilities. Messerschmidt et al. defines a performance measure for a tic-tac-toe game as follows:

$$S = (w1 - 58.8) + (w2 - 28.8) \quad (1)$$

where w1 and w2 correspond to the percentage of games won as player one and player two respectively. Stronger players will have higher S values, and weaker players will have S values closer to 0. It is possible to achieve negative S values, which point to players learning 'how to lose'. In the experiments conducted negative learning did not occur. A confidence coefficient of 90% is used when statistically computing the confidence interval for the games played.

5.3 Neural network configuration and input pattern creation

A standard 3-layer feed forward neural network architecture is implemented, with the input layer consisting of 9 input nodes (with an additional bias unit), a hidden layer of varying size (between 3 and 15 hidden nodes with an additional bias unit – depending on the simulation) and a single output unit. Sigmoid evaluation functions are used in each of the neurons. All the weights of the neural network are randomly instantiated in the range (-1/sqrt(fanin) , 1/sqrt(fanin)) [Engelbrecht 2002], where *fanin* represents the number of incoming weights to a specific neuron.

The 3x3 matrix board state is represented as an array of 9 values. A value of 1.0 is assigned for a piece/symbol belonging to the player busy with the decision making process, a value of 0.0 for opponent pieces/symbols and a value of 0.5 for an empty space on the board. This scheme will result in identical board states to be inversely represented depending on whether the player played first or not. The array of 9 values is used as input to the neural network. The numeric output of the network corresponds to the 'ranking' of the proposed move, with higher rankings indicating better moves and is used as an evaluation of the leaf nodes of a game tree.

This approach differs from the one Messerschmidt et al. implemented, where 1.0 represents a cross, 0.0 a circle and 0.5 an open space. He included an additional input unit to the network that provided the player number (0 or 1). Using this scheme, an identical board state's representation for both players will differ only by a single node (the player number input).

5.4 Particle swarm configuration

As already discussed, each particle consists of an n-dimensional weight vector that corresponds to the weights of a neural network. Each particle can be seen as a single game playing agent in a population of players, with the substitution of particle information (weight values) into a neural network framework resulting in unique board evaluations for that specific player.

Due to the need of the PSO algorithms to compare performance of the current particle against its personal best performance thus far, each particle's personal best configuration is duplicated and added as an additional player in the population of players. This personal best particle also competes in the training process, but only serves as a performance comparison and does not get altered – only replaced if it is outperformed.

For the experiments, a fixed set of PSO variables are used in order to compare the results with previous work conducted by Messerschmidt et al. [2002]. The c1 and c2 variables are both set to 1.0, while the inertia weight is also fixed at 1.0 (no influence). One of the major differences between the algorithm comparisons and previous studies is the exclusion of the maximum velocity parameter. For the lBest, gBest and Von Neumann architecture tests the particles' velocities were not constrained.

Some tests were performed to try and optimise these baseline configurations using other values for the c1, c2, inertia weight and vMax parameters, with adequate success.

6. RESULTS

Experiments were conducted testing a variety of population and hidden layer sizes. In addition to the standard particle swarm algorithms tested, the new Von Neumann neighbourhood structure is used for the first time to train game learning agents. The performance measure defined in section 6.2 is used to calculate each configuration's ability to beat a random player. Higher s-values are preferred. Each s-value also lists an accompanying confidence interval calculated using a confidence coefficient of 90% from the t-distribution. The values are calculated over 30 simulation runs each.

Table 1 lists the performance of an agent trained through co-evolution by the local best PSO algorithm when played against a player making random moves. The performance of the global best PSO algorithm is listed in table 2. It is obvious from visual inspection that the gbest algorithm performs worse than lbest, which corresponds to existing trends in PSO research.

The performance of the Von Neumann neighbourhood structure is listed in table 3, with very positive results. Von Neumann outperforms lbest and gbest in the majority of test cases, while matching lbest's performance for the remainder of the tests.

Figure 1 compares the standard PSO algorithms (with vMax disabled) against the performance of PSO and evolutionary programs (EP) tested by Messerschmidt for neural networks with 7 hidden units. The PSO algorithms with no maximum velocity restriction significantly outperforms Messerschmidt et al.'s PSO and EP configurations in every single case, with the Von Neumann algorithm once again proving superior in the majority of tests.

Figure 2 indicates the performance gain that can be achieved through customising standard PSO variables for each specific particle swarm size. In this specific instance the values used to compare against Messerschmidt et al. is changed in the following way: c1 = 1.0, c2 = 2.5 and vMax = 170. The inertia value remains unchanged. The tests were conducted on a swarm of 5 particles. The 'optimised' values result in significant performance increases in most cases, as can be more clearly seen by the polynomial trend-lines for each graph.

Lastly, figure 3 illustrates the convergence properties of PSO while training a neural network. The average weight values stabilise for prolonged periods, indicating that a satisfactory optimum has been reached.

Particles	Hidden Units							Average
	3	5	7	9	11	13	15	
5	24.1383 ±0.0020261	27.1410 ±0.0020046	28.5990 ±0.0020030	28.7833 ±0.0020004	29.2807 ±0.0019927	27.5753 ±0.0020050	25.0897 ±0.0020222	*27.229614 ±0.002008*
10	32.9270 ±0.0019521	28.6597 ±0.0019904	30.9273 ±0.0019737	29.1230 ±0.0019787	30.2013 ±0.0019795	28.5807 ±0.0019960	29.4373 ±0.0019964	*29.979471 ±0.001981*
15	33.6943 ±0.0019568	28.7860 ±0.0020141	30.1543 ±0.0019880	33.6173 ±0.0019529	28.6750 ±0.0019985	30.0497 ±0.0019860	30.6863 ±0.0019742	*30.808986 ±0.001981*
20	28.8050 ±0.0019891	30.2670 ±0.0019823	31.1017 ±0.0019667	31.3707 ±0.0019760	33.1450 ±0.0019504	33.0493 ±0.0019648	33.8870 ±0.0019532	*31.660814 ±0.001969*
25	29.5123 ±0.0019962	32.2863 ±0.0019609	30.4833 ±0.0019811	28.6087 ±0.0020060	29.6403 ±0.0019921	34.7610 ±0.0019498	29.1687 ±0.0019862	*30.637229 ±0.001982*
30	26.9153 ±0.0020053	28.9407 ±0.0019883	31.8077 ±0.0019745	28.1220 ±0.0019961	30.2430 ±0.0019825	30.5983 ±0.0019852	31.9637 ±0.0019632	*29.798671 ±0.001985*
35	31.5077 ±0.0019817	29.4867 ±0.0019878	30.7610 ±0.0019781	25.9843 ±0.0020192	29.3933 ±0.0019964	31.7043 ±0.0019812	28.4090 ±0.0019957	*29.606614 ±0.001991*
40	28.5287 ±0.0019938	30.0900 ±0.0019719	31.0027 ±0.0019855	30.2270 ±0.0019914	27.6470 ±0.0020005	30.7377 ±0.0019803	31.8813 ±0.0019683	*30.016343 ±0.001985*
45	28.5630 ±0.0019938	27.4617 ±0.0020127	27.8940 ±0.0020053	27.4003 ±0.0020025	33.5873 ±0.0019589	30.8893 ±0.0019802	26.9137 ±0.0020142	*28.958471 ±0.001995*
50	31.9237 ±0.0019516	30.2567 ±0.0019842	29.1200 ±0.0019969	26.7167 ±0.0020120	32.6270 ±0.0019584	31.3827 ±0.0019703	30.6550 ±0.0019830	*30.383114 ±0.001979*
Average	*29.6515 ±0.0019847*	*29.3376 ±0.0019897*	*30.1851 ±0.0019853*	*28.9953 ±0.0019935*	*30.4440 ±0.0019810*	*30.9328 ±0.0019799*	*29.8092 ±0.0019857*	

Table 1: Performance of lBest to train a superior agent when compared to a random player.
Higher S-values preferred.

Particles	Hidden Units							Average
	3	5	7	9	11	13	15	
5	23.8750 ±0.0020368	18.5720 ±0.0020620	31.3060 ±0.0019901	22.7353 ±0.0020488	18.0367 ±0.0020508	28.1683 ±0.0020131	21.0053 ±0.0020445	*23.385514 ±0.002035*
10	24.6860 ±0.0020383	20.0157 ±0.0020451	30.0207 ±0.0019964	30.4213 ±0.0019873	26.0243 ±0.0020090	25.7027 ±0.0020484	21.8407 ±0.0020435	*25.530200 ±0.002024*
15	29.4563 ±0.0019968	25.8220 ±0.0020174	29.7477 ±0.0019952	27.9913 ±0.0020088	28.8753 ±0.0019956	25.6050 ±0.0020106	22.5307 ±0.0020430	*27.146900 ±0.002010*
20	26.8647 ±0.0020034	27.7497 ±0.0020147	27.4233 ±0.0020093	27.2117 ±0.0020100	29.4540 ±0.0019946	30.1043 ±0.0019792	26.1790 ±0.0020288	*27.855243 ±0.002006*
25	25.5083 ±0.0020189	29.4643 ±0.0020004	24.1423 ±0.0020269	28.6997 ±0.0019922	26.0070 ±0.0020100	31.5457 ±0.0019617	29.0127 ±0.0019983	*27.768571 ±0.002001*
30	22.2600 ±0.0020400	22.8560 ±0.0020431	25.3557 ±0.0020196	27.9470 ±0.0020001	27.6160 ±0.0020083	23.5457 ±0.0020192	28.4513 ±0.0019960	*25.433100 ±0.002018*
35	21.9727 ±0.0020236	30.9743 ±0.0019908	27.8827 ±0.0020142	26.0450 ±0.0020111	24.1777 ±0.0020323	28.5027 ±0.0019965	31.1313 ±0.0019743	*27.240914 ±0.002006*
40	27.7207 ±0.0020026	25.5043 ±0.0020168	24.1233 ±0.0020177	29.0493 ±0.0019964	26.6637 ±0.0020113	29.5673 ±0.0019902	26.8940 ±0.0020155	*27.074657 ±0.002007*
45	30.0580 ±0.0019931	28.0940 ±0.0019984	25.3267 ±0.0020149	32.3543 ±0.0019652	29.3117 ±0.0019898	32.4480 ±0.0019755	27.3703 ±0.0020096	*29.280429 ±0.001992*
50	25.8290 ±0.0020244	25.2800 ±0.0020209	26.7350 ±0.0020011	29.0033 ±0.0019974	24.6987 ±0.0020262	27.9267 ±0.0020169	22.5030 ±0.0020428	*25.996529 ±0.002019*
Average	*25.8231 ±0.0020178*	*25.4332 ±0.0020210*	*27.2063 ±0.0020085*	*28.1458 ±0.0020017*	*26.0865 ±0.0020128*	*28.3116 ±0.0020011*	*25.6918 ±0.0020196*	

Table 2: Performance of gBest to train a superior agent when compared to a random player.
Higher S-values preferred.

Particles	Hidden Units							Average
	3	5	7	9	11	13	15	
15	31.0567 ±0.0019773	28.7540 ±0.0019930	31.3130 ±0.0019753	31.9187 ±0.0019700	32.2927 ±0.0019630	28.3747 ±0.0020004	29.3333 ±0.0019818	*30.434729 ±0.001980*
20	29.1440 ±0.0019900	29.2823 ±0.0019883	34.8490 ±0.0019529	33.3323 ±0.0019551	33.4453 ±0.0019535	32.3253 ±0.0019588	32.0373 ±0.0019644	*32.059357 ±0.001966*
25	30.0717 ±0.0019822	30.0000 ±0.0019889	32.6310 ±0.0019577	32.2117 ±0.0019536	31.0423 ±0.0019731	33.5230 ±0.0019642	31.0607 ±0.0019762	*31.505771 ±0.001971*
30	28.5637 ±0.0020083	30.8887 ±0.0019741	31.6957 ±0.0019768	28.6523 ±0.0020050	31.5673 ±0.0019726	29.1123 ±0.0019990	26.5627 ±0.0020167	*29.577529 ±0.001993*
35	32.6300 ±0.0019643	31.0873 ±0.0019797	27.6457 ±0.0020032	27.9657 ±0.0020028	29.1573 ±0.0019875	31.6967 ±0.0019694	28.4807 ±0.0020086	*29.809057 ±0.001988*
40	29.3983 ±0.0019942	28.9783 ±0.0019901	30.2877 ±0.0019783	27.3030 ±0.0020030	29.9420 ±0.0019863	29.3197 ±0.0019998	32.7027 ±0.0019560	*29.704529 ±0.001987*
45	29.9777 ±0.0019820	29.6120 ±0.0019873	29.6120 ±0.0019873	29.7713 ±0.0019945	29.7160 ±0.0019987	26.8357 ±0.0019999	27.8463 ±0.0020072	*29.053000 ±0.001994*
50	24.8740 ±0.0020124	26.3273 ±0.0020253	30.9993 ±0.0019934	30.7183 ±0.0019865	28.9213 ±0.0019955	32.3717 ±0.0019681	26.4413 ±0.0020062	*28.664743 ±0.001998*
Average	*29.4645 ±0.0019888*	*29.3662 ±0.0019908*	*31.1292 ±0.0019781*	*30.2342 ±0.0019838*	*30.7605 ±0.0019788*	*30.4449 ±0.0019825*	*29.3081 ±0.0019896*	

Table 3: Performance of Von Neumann PSO algorithm to train a tic-tac-toe game-playing agent. Higher S-values preferred.

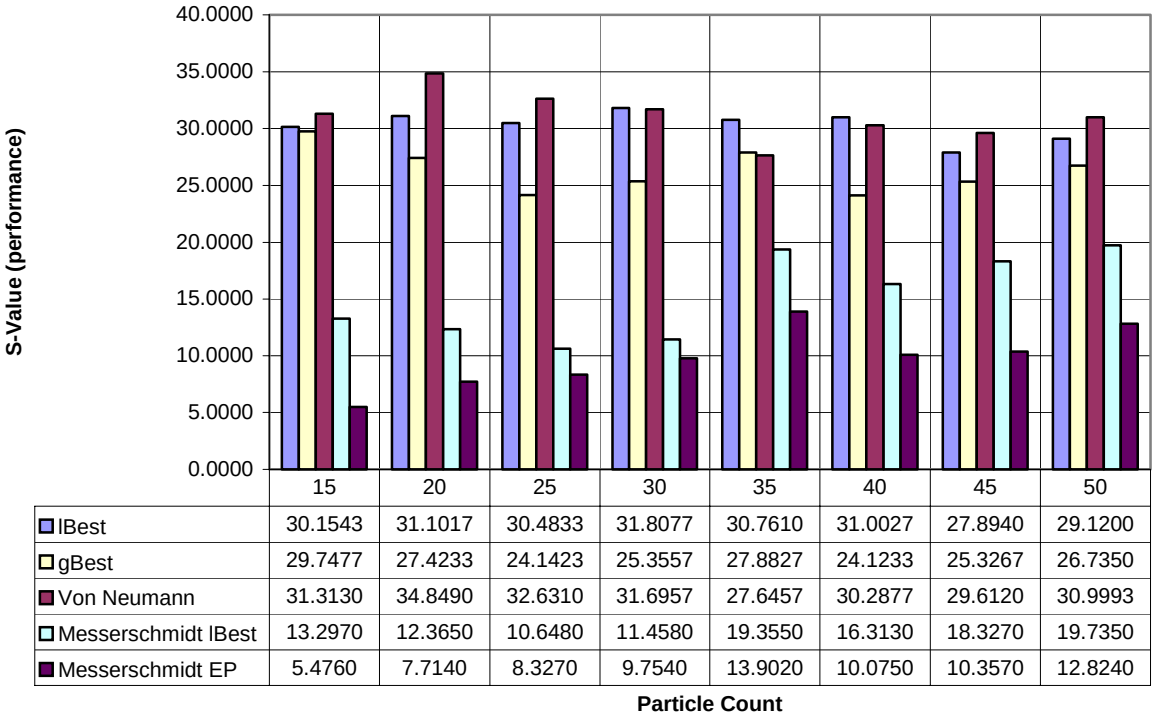

Figure 1: Comparing performance of standard PSO architectures against existing research using neural networks with 7 hidden units.

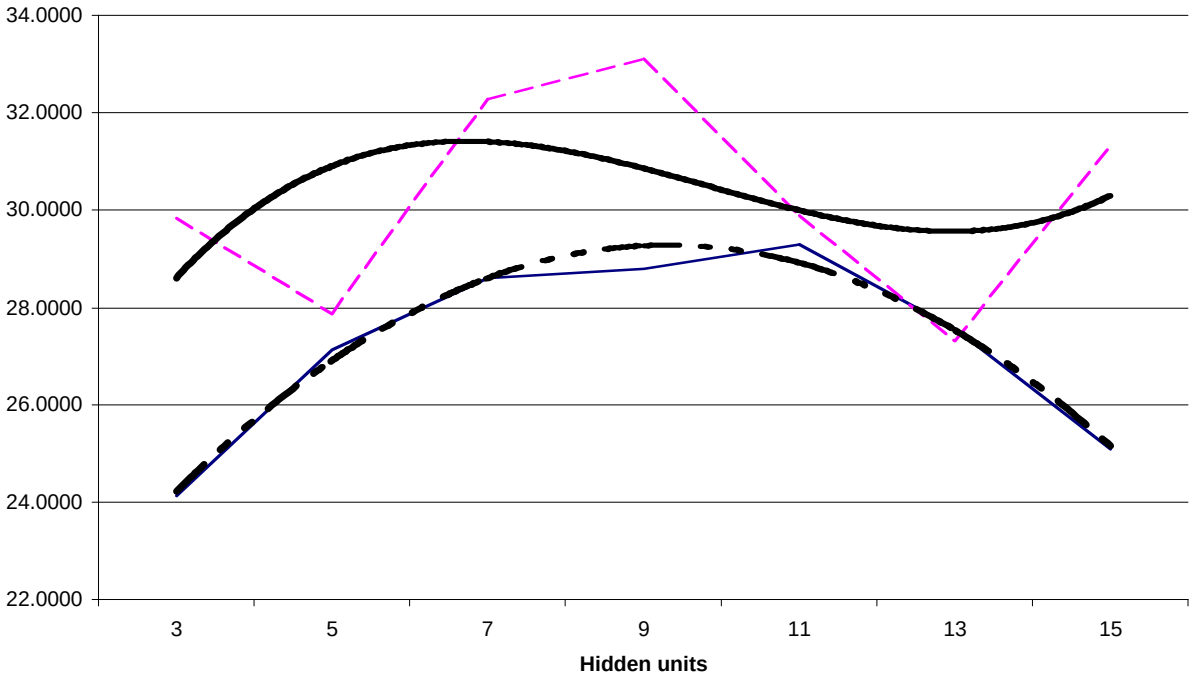

Figure 2: Increased performance with custom PSO variable setup.
Tests conducted using the lBest PSO algorithm and 5 particles.

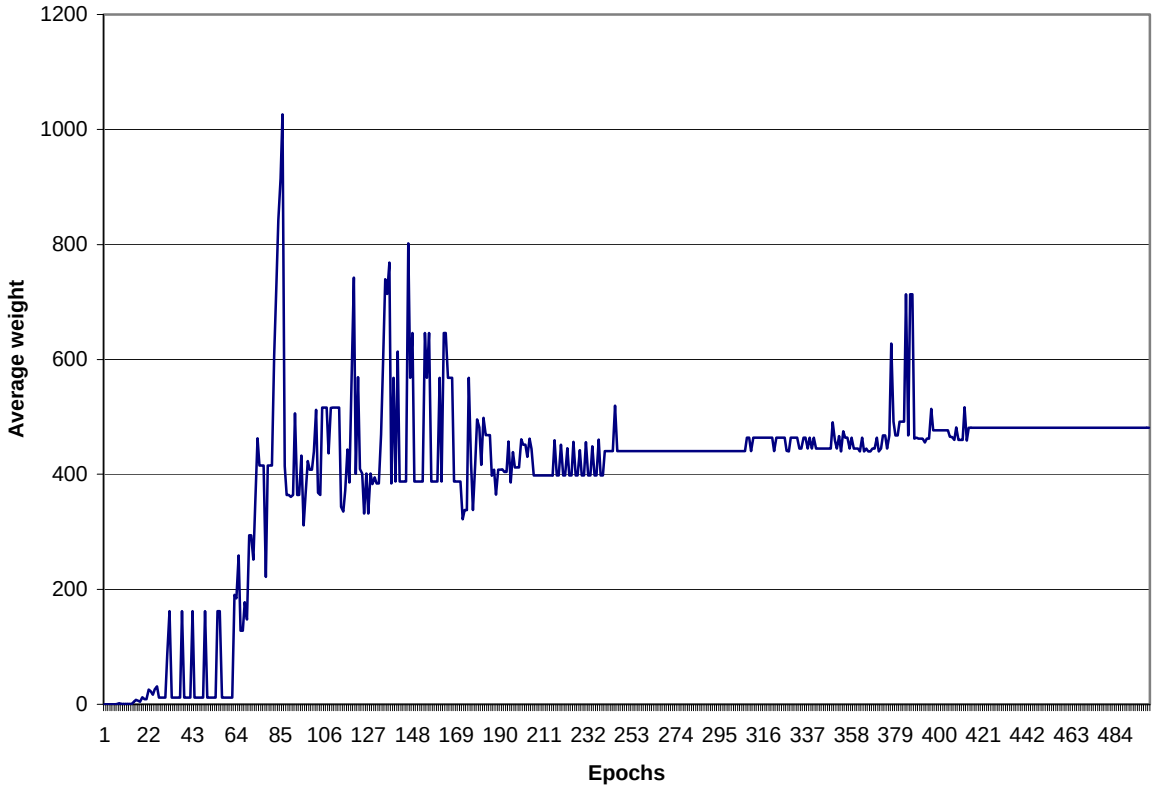

Figure 3: Convergence behaviour of PSO in modifying NN weights.

7. CONCLUSION AND FUTURE WORK

This paper has shown that PSO can successfully train neural networks to evaluate competitive board states during a tic-tac-toe game, without the use of any prior strategic game knowledge apart from its official rules. It has also shown how co-evolution can be used to drive this training process. Judging by the positive results obtained through the experimental work, a more challenging problem like checkers warrants further investigation. The performance of PSO in training co-evolving game playing agents in a much larger problem space will once again be compared to existing game learning research. The use of niching algorithms to construct game playing strategies will also be thoroughly investigated.

8. ACKNOWLEDGMENTS

The financial assistance of the National Research Foundation towards this research is hereby acknowledged. Opinions expressed in this paper and conclusions arrived at, are those of the authors and not necessarily to be attributed to the National Research Foundation.

9. REFERENCES

CHELLAPILLA, K. FOGEL, D. 1999. Evolving neural networks to play checkers without expert knowledge. IEEE Transactions on Neural Networks. 10(6):1382-1391.

CHELLAPILLA, K. FOGEL, D. 2000. Anaconda Defeats Hoyle 6-0: A Case Study Competing an Evolved Checkers Program against Commercially Available Software. In Proceedings of Congress on Evolutionary Computation, July 16-19 2000, La Jolla Marriot Hotel, La Jolla, California, USA, 857-863.

CHURCHILL, J. CANT, R. AL-DABASS, D. 2001. A new computational approach to the game of Go. In Proceedings of the 2nd Annual European Conference on Simulation and AI in Computer Games (GAME-ON-01). London. 2001. 81-86.

DE COSTE, D. 1997. The future of chess-playing technologies and the significance of Kasparov versus Deep Blue. In Deep Blue Versus Kasparov: The Significance for Artificial Intelligence: Papers from the 1997 AAAI Workshop, pages 9-13. AAAI Press. Technical Report WS-97-04.

ENGELBRECHT, AP. 2002. Computational Intelligence: An Introduction. Wiley and sons.

FOGEL, D. 2001. Blondie24: Playing at the edge of A.I. Morgan Kaufmann Publishers.

KENDALL, G. WHITWELL, G. 2001. An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In proceedings of CEC2001 (Congress on Evolutionary Computation 2001), COEX Center, Seoul, Korea, May 27-29, 995-1002.

KENNEDY, J. EBERHART, RC. 1995. Particle Swarm Optimization. Proceedings of the IEEE International Conference on Neural Networks, Perth, Australia. Vol IV, 1942-1948.

KENNEDY, J. MENDES, R. 2002. Population Structure and Particle Swarm Performance. In Proceedings of Congress on Evolutionary Computation (CEC 2002), Honolulu, Hawaii USA.

MESSERSCHMIDT, L. ENGELBRECHT, AP. 2002. Learning to play games using a pso-based competitive learning approach. In Proceedings of the 4th Asia-Pacific Conference on Simulated Evolution and Learning, Singapore.

NILSSON, N. 1998. Artificial Intelligence: A new synthesis. Morgan Kaufmann Publishers.

REINEVELD, A. 1989. NegaScout - A Minimax Algorithm faster than AlphaBeta. http://www.zib.de/reinefeld/nsc.html. Accessed on 22 May 2003.

SAMUEL, A. 1959. Some studies in machine learning using the game of checkers. IBM Journal of Research and Development. 3(3):211-229.

SCHAEFFER, J. et al. 1992. A world championship calibre checkers program. Artificial Intelligence, Vol. 53, No. 2-3, pp. 273-290

SCHAEFFER, J. 2001. The Games Computers (and People) Play. Academic Press, Vol. 50 (ed Zelkowitz M.V.), 189-266.

VAN DEN BERGH, F. ENGELBRECHT, AP. 2001. Training Product Unit Networks using Cooperative Particle Swarm Optimisers. Proceedings of the IEEE International Joint Conference on Neural Networks, Washington DC, USA. 126-132.