Co-evolution versus Self-play Temporal Difference Learning for Acquiring Position Evaluation in Small-Board Go

Thomas Philip Runarsson Member, IEEE, and Simon M. Lucas Member, IEEE,

Abstract— Two learning methods for acquiring position evaluation for small Go boards are studied and compared. In each case the function to be learned is a position-weighted piece counter and only the learning method differs. The methods studied are temporal difference learning using the self-play gradient-descent method and co-evolutionary learning, using an evolution strategy. The two approaches are compared with the hope of gaining a greater insight into the problem of searching for “optimal” zero-sum game strategies.

Using tuned standard setups for each algorithm, it was found that the temporal-difference method learned faster, and in most cases also achieved a higher level of play than co-evolution, providing that the gradient descent step size was chosen suitably. The performance of the co-evolution method was found to be sensitive to the design of the evolutionary algorithm in several respects. Given the right configuration, however, co-evolution achieved a higher level of play than temporal difference learning.

Self-play results in optimal play against a copy of itself. A self-play player will prefer moves from which it is unlikely to lose even when it occasionally makes random exploratory moves. An evolutionary player forced to perform exploratory moves in the same way can achieve superior strategies to those acquired through self-play alone. The reason for this is that the evolutionary player is exposed to more varied game-play, because it plays against a diverse population of players.

I. INTRODUCTION

Games provide competitive dynamic environments that make ideal test-beds for computational intelligence theories, architectures, and algorithms. Natural evolution can be considered a game, where the rewards for an organism that plays a good game of life are the propagation of its genetic material to its successors. Natural evolution is co-evolution, where the fitness of an individual is defined with respect to its competitors and collaborators, as well as the environment.

Many human-designed games are much simplified analogs of real-world situations, though staggering complexity can arise from a few simple rules. Alan Turing proposed chess as a test-bed for artificial intelligence, and is believed to have devised the first chess-playing computer program, which he simulated by hand, and which played very poorly. In 1950, Shannon published the first paper outlining various types of chess-playing algorithms, and proposed what was essentially a hill-climber or (1 + 1) evolution strategy (ES) that could improve its play by monitoring its own performance.

While the performance of computer players has now exceeded human grandmaster level in games such as chess and checkers, Go has proved to be a much tougher game, and the best computer players now play at the level of a good novice. While Shannon may have been the first to suggest a kind of evolutionary algorithm for a self-improving chess player, temporal difference learning (TDL) was applied by Samuel as far back as 1957 [2] and Michie in 1961 [3].

This leads us to the subject of the paper: a comparison between TDL and co-evolutionary learning (CEL) for game state evaluation in 5 × 5 Go. This kind of comparison seems timely, since in the last few years, there has been a surge of interest in evolutionary approaches to learning game state evaluation functions. Much of this was probably inspired by the work of Pollack and Blair [4], and Chellapilla and Fogel [5], [6], [7]. These papers showed that simple evolutionary algorithms, when applied in a co-evolution framework could achieve surprisingly high levels of play in backgammon and checkers (also known as draughts), respectively. So far, we are aware of only one other direct comparison of TDL versus CEL, by Kotnik and Kalita [8], for the game of rummy. In that study they found that CEL outperformed TDL, but they used TDL with no intrinsic exploration, relying instead on the randomness of the cards. They also initialised the TDL networks with random weights. In our experience, either or both of these choices can lead to poor TDL performance. Whether or not the randomness of the cards alone can force sufficient exploration of game space is an open question.

Darwen [9] also compared CEL with TDL, but his comparison was less direct. Darwen showed that using co-evolution, he could evolve a linear backgammon position evaluator that outperformed the linear version of Pubeval [10], a publicly available position evaluator found using TDL. This comparison did not address the computational effort involved in each approach, however, and there may also have been other experimental details that differed. Nonetheless, it is noteworthy that CEL was in this case able to find a different and better player from TDL.

While it is interesting to investigate what can be learned with evolutionary algorithms, without any addition of human knowledge, both TDL andCEL are capable of bootstrapping in this way. It is therefore important to investigate which method works best on a range of games, and a range of game-state evaluation architectures, and why? While this paper is restricted to learning the parameters of a position-weighted piece counter for small board Go, the plan is to extend the investigation to other games and to other architectures in the future.
The rest of this paper is structured as follows. The next section discusses the background to this work and outlines the main issues to consider when learning game playing strategies. Section III gives a brief description of the game of Go. Section IV describes the common aspects of the TDL and CEL experiments and defines the position evaluation function that each method will attempt to learn. Section V describes the setup and results of the TDL system, while section VI reports the CEL experiments. Further discussion on the experimental results is presented in section VII, and section VIII concludes the paper.

II. LEARNING GAME POSITION VALUE FUNCTIONS

This section discusses the main issues to consider when setting up experiments to learn a game strategy. This paper studies small-board (5 × 5) Go, for reasons discussed in the next section. Having chosen the game, one then needs to decide:

- The learning algorithms: TDL and CEL for this paper.
- How the learning module will be interfaced to the game.
- The architecture of the learner (here a position-weighted piece counter).
- The game-playing setup: whether learning is through self-play (TDL) or co-evolution (CEL), or learning against a fixed set of opponents.
- How to ensure exploration during learning.
- How to choose the depth of ply when using minimax during learning.

Each of these issues is now explored in more detail.

A. TD Learning versus Evolutionary Learning

A policy is a mapping of game states to moves that should be taken when in each state. In TDL, which is one of the most successful and widely applied forms of reinforcement learning, this is achieved by estimating a value or utility function. This is the same heuristic function that is needed to estimate the outcome of a game given the game state. Moves are made that seek to maximize (or minimize) this value. The policy is based on the value function. The TDL method notices which game states are observed during play and uses this information to update the value function. This, together with a differentiable value function, results in a very efficient search. TDL includes a gradient descent step size, the setting of which is critical in order to achieve good performance. Still to this date the most celebrated success of TDL for game playing is TDGammon [11].

Unlike TDL, where the space of policies is searched indirectly by learning a value function, the evolutionary approach is to search the policy space directly. Furthermore, evolutionary learning does not usually take advantage of information available during the course of play and instead the merit of a policy must be evaluated by playing a number of games against an opponent. Having said this, it should be noted that the design of the fitness function is part of the skill in applying evolutionary algorithms to complex problems, and it is possible to design fitness functions that provide richer information than simply win, lose, or draw. With games such as Go, for example, one can also measure the degree of win by using the game score defined by the rules directly. Another alternative is to assign credit to good game states achieved during play, even if the player ultimately performed poorly. This was done by Stanley and Miikkulainen [12], for example, and there also exist hybrid forms of neuro-evolution [13], [14] that do attempt a form of credit assignment. Sophisticated fitness functions aim to provide more information to the evolutionary algorithm, and therefore speed up the search. They avoid the ‘needle in a haystack’ problem of measuring the performance of an evolved player against a vastly superior player. Hence, a gradient can be provided where one would otherwise not exist. The design of such fitness functions involves some human expertise, which may be incorrect, especially when being applied to rating an algorithmic player. Using incorrect information is likely to diminish the ultimate quality of the evolved player.

Generally speaking, evolutionary methods are relatively easy to apply. There are still many decisions to make, however, and in some cases the choices made can dramatically affect the degree of learning obtained. In general, the setup of an evolutionary algorithm includes the selection and replacement policy, the fitness function, the variation operators (mutation and/or recombination) and population size, together with any special features such as specification [15].

This paper shows that details such as the population size and the replacement policy (i.e., whether one uses a (µ, λ) or a (µ + λ) ES) have a profound effect on the performance of the evolved players.

Note that TDL and CEL aim to learn closely related, yet subtly, different things. TDL attempts to learn the probability of winning from each state or the expected future rewards, while CEL attempts only to learn a relative ordering on the set of game states. Thus, while the absolute values of the functions learned by each method will generally be different, the policies learned may or may not be the same. The reader is also referred to Moriarty et al. [16] for further reading on evolutionary versus reinforcement learning.

On a related theme, particle swarm optimization (PSO) has also been applied to self-acquired game learning [17], [18]. This is very similar to co-evolutionary learning in its focus on end results rather than credit assignment, except that the individuals in the population have velocity and exert direct influence on each other’s adaptations.

B. Interface Between Game Engine and Adaptive Players

There are at least two distinct ways in which both the temporal difference and evolutionary players may be interfaced to the game, namely, as a move selector or as a position evaluator. In reinforcement learning terms, these are known respectively as action-value and state-value schemes [19].

In the case of the move selector, the input to the player is the current state of the board, while the output from the player is the selected move. Fogel [20] used this approach when evolving a multi-layer perceptron to play tic-tac-toe (also known as noughts and crosses). Since the evolved player (at all stages of evolution, but especially early on) may have
great difficulty in playing a legal game, the move selections are typically filtered to select the best legal move. In Fogel’s approach, the tic-tac-toe network had an output for each board position, and the selected move was the legally playable square with the strongest corresponding output. Angeline and Pollack [21] attempted a more ambitious task where their system was encouraged to learn to play legal moves, the penalty for an illegal move being a missed turn.

The position evaluation interface provides a future board state as the input to the player, and the player returns a scalar evaluation of how favourable the board state is to that player. For example, strong positive might be interpreted as very good, strong negative as very bad. The move selection may then be made by picking a move that leads to the board state that was evaluated as being the most favourable. The choice of player interface has an important effect on how flexibly the player can be deployed. The position evaluator is of great practical value, since it may be deployed within a standard minimax algorithm. If a true evaluation function has been learned, then increasing the search depth used by the minimax algorithm will generally result in a more powerful player.

The move selection interface, on the other hand, has the advantage that the evolved player is in full control of the game. This is more satisfying from an evolutionary programming perspective: evolving a fully autonomous player that does not require any assistance from sophisticated hand-coded search algorithms is clearly a more impressive achievement than simply evolving the static evaluation function. Also, while in some games minimax search is very effective, in other games, especially those with large branching factors such as Go, minimax is of less value. In these cases, the fully evolved player potentially offers some advantages, in that it is able to specify the entire game playing strategy. Such an advantage has yet to be demonstrated in practice, however.

Tom English provided an evolved recurrent neural network player of this type (i.e., a player that did not rely on any minimax search) called Dynamo as an entry to the WCCI (IEEE World Congress on Computational Intelligence) 2002 Checkers Competition. All other entries were game position evaluation functions to be plugged into a minimax search engine. Dynamo was the leading entry when the minimax search was restricted to 1-ply, but was defeated soundly at any ply greater than this².

Most recent TDL and CEL approaches to game playing have focused on evolving a neural network or a weighted piece counter. The weighted piece counter (and of course, the unweighted piece counter) can also be implemented as a single-layer perceptron with a linear output.

For Go, the simplest and most direct approach is to have one input for each board position, with the input value being 0 for empty, +1 for Black, and −1 for White (or vice versa). Such a coding fails to exploit any natural invariances that the game may have, however. Many board game positions have some degree of invariance to rotation, reflection, and translation. Exploiting these invariances can lead to faster evolution, since the network only has to learn to play the first move in the corner is good (or bad), and does not have to learn this separately for each corner. However, this analysis assumes a perfectly rational opponent. A weak opponent may respond well to a first move in the top-right corner while playing a poor response to an opening move in the top-left corner.

In learning Go strategy, there has also been some work on graph collapsing [22]. This approach is based on the idea that a group of stones share a common fate, hence the term Common Fate Graph representation, where each group of stones maps to a single node of a graph. This allows a much more compact representation of board positions, one that aims to retain the most pertinent information present in the full board representation.

Schraudolph et al. [23] used an input processing layer with various size convolutional (or sliding) windows, together with symmetry pre-processing within their TD-Go system. Stanley and Miikkulainen [12] also used a sliding window approach in their ‘roving eye’ method. Enzenberger [24] used a neural network together with TDL where Go positions were simplified by segmenting the position into parts.

Kendall et al. [25] also used a convolutional approach (but without the symmetry considerations). The current work is more concerned in the comparison of CEL versus TDL, and adopts a direct input scheme. Note that when choosing a direct input scheme, there are still possible variations. The simplest, as mentioned above, is to code each position as a single scalar value. It is also possible to have two inputs for each board position, one for each player. If the evaluation function is coding for the presence of certain features, such as lines in tic-tac-toe, for example, this makes it easier for the network to distinguish important cases, and enables direct discrimination between a line with one piece of each color on it, and a completely empty line.

For more conventional AI approaches that involve hand-written heuristic evaluation functions, there has been much work on designing and tuning these functions. The reader is referred to Fotland [26], and Muller [27], for comprehensive discussion of this topic. Note that these human-designed functions are unlikely to be optimal, and could therefore most likely benefit from some automated tuning procedure. In general this would be more easily accomplished using CEL rather than TDL, since with CEL there is no requirement for the evaluation function to be differentiable.

Finally, one of the most sophisticated representation schemes used for a trainable game player is the one developed by Enzenberger [28] [24] for his NeuroGo system. NeuroGo, now in version 3, used low-level features based on a stone’s position on the board, as well as various high-level features. These included features based on sets of connected stones, and also some fairly sophisticated relationships between sets

²See http://algoval.essex.ac.uk/ace/algoval/wcci/CompetitionResults.pdf
of connected stones, such as the shortest path distance between them via empty intersections. Enzenberger’s system elegantly captures board symmetries by the very nature of the features it extracts. Evidence of the merit of this approach can be seen in NeuroGo’s performance in the computer olympiads over the last several years, with it achieving second place (out of nine entries) for 9 × 9 Go in 2003.

In general, more sophisticated input coding schemes are expected to benefit both TDL (where applicable) and CEL methods, though whether they benefit one method more than the other is an open question.

D. Absolute or Relative Fitness Measures

A key component of any evolutionary algorithm is how the fitness of an individual is measured. Evolved game players can either be measured by how well they play against a fixed opponent, or by how well they play against each other — the latter case is known as co-evolution within evolutionary learning, or self-play within temporal difference learning.

Both co-evolution and self-play are especially interesting, since they support the idea of learning game strategy without providing any human expertise. The absolute fitness measure, however, may limit the maximum ability of the learned players to play just strongly enough to defeat the hand-designed player. It is also possible to use a combination of absolute and relative fitness measures.

While there are many possible problems with co-evolution, such as cycling, forgetting, lack of diversity, and loss of gradient, Ficici [29] argues that these are all symptoms of a single underlying cause: the lack of a suitable solution concept. Much of the work on co-evolving game strategies has not been too concerned with these potential problems, and indeed some of these problems may be more relevant when applying co-evolution to heterogeneous populations [30]. However, Ficici does report that many of these potential difficulties can be overcome by ensuring that the population is sufficiently diverse. Our results in section VI accord with this, and demonstrate that a larger population, together with enforced diverse behaviour via partially random play is the key to making CEL compete with TDL for our problem setup.

One promising idea that appears to have not been tried, is the combination of TDL and CEL in the sense of using TDL’s credit assignment strategy, together with CEL-style population dynamics. This may lead to more realistically varied play than can be achieved with TDL self-play alone.

Lubberts and Miikkulainen [31] compared co-evolution with evolution against a fixed player, also for 5 × 5 Go, and found that co-evolution significantly outperformed evolution against a fixed player. For their fixed player they used an earlier version of GNUGo than the one used here, so the result that they evolved players able to beat GNUGo does not apply to the current version of GNUGo, which is significantly stronger.

This paper (see section V) finds similarly, that TDL with self-play significantly outperforms TDL against a fixed opponent.

E. Exploration and Noisy Fitness Evaluation

The aim of the fitness evaluation function should be to determine the playing quality of the current individual. This would be done ideally by playing against as many players as possible, and as many times as possible. In play between humans, the same players may play many times, with each game being very different. When evaluating the relative quality of two deterministic players, A and B, however, a problem arises. Unless care is taken to overcome it, only two outcomes would be observed: when player A plays first, and when player B plays first. This is irrespective of how many games are played. There are three possible solutions:

- Force opening moves.
- Measure the fitness of A and B against a separate random (or partially random) player.
- Introduce randomness into the play of A and / or B.

Each of these methods has strengths and weaknesses. Forcing opening moves is a simple option, though in some games this can force disastrous moves on players that would normally have sufficient knowledge to avoid them.

Another solution is to use an absolute fitness measure (e.g., performance against GNUGo) — against an opponent capable of randomly varied play. In the case of Go, an elegant variation on this theme is to measure the handicap you have to give to a pure random player before it beats you.

There are two main ways to enforce random play — either based on the absolute value of position evaluations (and hence moves), or the rank of a move when sorted into best-first order. This is analogous to the design of a selection function within an evolutionary algorithm.

In the first case, a simple solution is to add a small amount of noise to the network output for each position evaluation, though careful judgements are needed as to what constitutes small. A principled way of achieving this is to use a Gibbs distribution, which is one of the standard approaches used in TDL. In this case, the value of each move is put through a softmax function with a particular gain. In the case of a small gain, all moves will be equally likely. In the case of a large gain, the move with the highest value is sure to be picked; varying the gain allows any policy between these two to be adopted. During the course of evolution, the gain could start small, allowing a wide variety of moves to be explored, while the gain might be increased towards the end, favouring selection of the high-rated moves. Still, the value of the gain parameter, or the schedule of values needs to be selected empirically, and is yet another parameter to tune. The approach adopted in this paper is the $\epsilon$-greedy method which is an effective and popular means of balancing exploration and exploitation in TDL [19]. The only drawback with this method is that it is just as likely to choose the worst move as is the next best.

The second alternative, with a very intuitive interpretation, is the rank-based move selector. The approach here is to rank all the board positions (and hence possible moves) using the noise-free board evaluation function, then pick a move from the ranked list using an exponential distribution. This is achieved easily by working down the list starting at the
top, picking the current item with a probability of 0.9 (for example). This means that 90% of the time the best move will be picked, but this would be enough over the course of the game to ensure that every game was different in practice (if the end of the list has been reached, then the current move, i.e. the last one in the list, is selected).

In the case of games with a natural random element, such as backgammon [4], it is commonly assumed that sufficient variation in play is enforced by the roll of the dice or the shuffle of the cards, although it seems to us that this is not necessarily the case.

Variation in play is extremely important. As pointed out by Richards et al. [13], when evolving against a deterministic opponent, it is possible to evolve a particular sequence of moves that defeat it, rather than learning a general strategy.

On the other hand, when dealing with a population of players, the average performance of a player against a set of opponents can be used. Provided that the population is sufficiently large and diverse, then this should lead to a good evaluation. Note, however, that maintaining diversity within a population can be a significant problem in itself, and enforced random play can help to solve this.

Whichever measure is chosen, it is important that fitness should be some aggregate measured over many varied games. Without this, the fitness evaluation function will be too noisy, and while some evolution may still occur, the evolutionary process is always in danger of drifting away from good regions of the solution space.

F. Choice of Architecture to Evolve

The most common choice of architecture to evolve for position evaluation functions has been the multi-layer perceptron (MLP). The advantages of this architecture are two-fold: an MLP is in theory able to represent any continuous measurable functional mapping, and in practice, MLPs have been found to be quite evolvable.

There are still details that need to be chosen, such as the number of hidden layers, and the number of units in each layer. Although in theory, one hidden layer is sufficient to represent all continuous measurable functional mappings, more structured architectures can lead to much more concise representations of approximately equivalent mappings [32]. These details can be chosen by the human designer, or by evolution. A related point is the presence of any special convolutional features of the network. Where these exist, they are usually designed by hand. Where they are used only in the input layer of the network, they can be seen as features of the network architecture, or features of the input coding scheme.

In the simplest case of a network with no hidden layers, the network output is a weighted linear function of its inputs. This function has some important advantages over other more complex networks. Firstly, due to its simplicity, it can be evaluated more quickly than more complex architectures. Secondly, due to its linear nature, it can be evaluated incrementally. That is, when a board state changes, only the effects of the changes to the board need to be evaluated, making it even faster. This greater speed can be utilised in two ways. During evolution, it allows greater search ply to be used, and it allows more fitness evaluations to be performed within a given amount of CPU time. Then, during play, the greater speed allows significantly greater search depth. The downside is that it is incapable of learning anything of great complexity, unless specialized input features are chosen.

Hughes [33] successfully evolved a weighted piece difference function for a checkers player called Brunette, in deference to Chellapilla and Fogel’s Blondie24 [5] [6] [7]. When playing games where each player is allowed the same amount of CPU time per move, and allowed to search as deep as time permitted (up to about 8-ply), Blondie24 and Brunette appear to play at a similar standard against each other3, though differences in the minimax algorithms used by each player are also a factor that should be considered. While Blondie24 has acquired a more powerful evaluation function, the simplicity of Brunette’s evaluator allows many more game positions to be evaluated. Note that Chellapilla and Fogel [6] did show that their evolved neural network was superior to an unweighted piece counter, both fixed-ply and fixed CPU time, but a direct comparison has yet to be made with a weighted piece counter. A weighted piece counter can play a significantly stronger game than its unweighted counterpart for the same ply depth, and is nearly as fast to compute.

Having chosen an architecture, it must also be encoded for evolution. This is not an issue for TDL, which only learns the parameters of a fixed architecture. Likewise, the simplest approach when using evolution is to fix the architecture and evolve only the parameters. The problem with this is that fixing the architecture limits what can be learned. In his work on Tic-Tac-Toe, Fogel [20] allowed the number of hidden nodes to evolve, but otherwise kept the architecture fixed.

The approach of Stanley and Miikkulainen [12] [34] offers a solution to this. The idea is to start with a simple network and complexify it by adding hidden units. Although this works with a direct representation of the architecture, the fact that it evolves architectures of arbitrary complexity marks it out from the simplest form of direct encoding where the network architecture is entirely fixed, and only the weights are evolved. Due to the way hidden nodes are added, arbitrary feed-forward network topologies can be evolved.

In other areas of neural network evolution, there has been a significant interest in indirect coding schemes [35] [36] [37], but such coding schemes have not yet been applied to evolving game strategies.

One of the most interesting game playing neural architectures is the NeuroGo system of Enzenberger [28] [24] (also mentioned in Section II-C). NeuroGo uses an innovative neural architecture where the number of neurons and their interconnections grow or shrink depending not only on the size of the board, but also on the state of the board. NeuroGo also exploits dynamically computed weights, an example of this being the shortest path graph features used to connect some layers of the network. NeuroGo also uses weight types, where the value of a weight is common to all connections of

3 In independent personal communications from Evan J. Hughes and David B. Fogel.
a given type. This allows networks trained for one board size to be applied directly to another, typically larger size board (or at least used as a starting point for further training).

G. Ply Depth

Selecting a ply depth during learning is an interesting issue for both TDL and CEL. On the face of it, one might suppose that a good evaluation function would be good at any depth. This might lead to choosing 1-ply search during learning, i.e., consider only the effects of all possible immediate moves. This is the fastest alternative, and adopting this approach means that many more games can be played within the same time limit. Franken and Engelbrecht used 1-ply search in their PSO approach [17]. Fogel and Chellapilla [6] and Hughes [33] used 4-ply search when evolving their checkers players, as did Fogel et al. [14] for chess. Given that the time taken to play each game grows exponentially with the search ply depth, what reason lies behind this? Assuming that a real advantage does indeed exist for learning at greater ply (if the extra time taken is ignored), there are two reasons why this could be so. Firstly, the policy is tested on situations that are more likely to occur during actual play (since players evolved at 1-ply would not play a strong game if actually played competitively at 1-ply). Secondly, due to the greater expansion of the search tree, the evaluation function is applied many more times during each game, leading to a more thorough assessment of its strengths and weaknesses. Note also that in practice, depending on the game engine used, the search ply may vary in response to the state of the game. For example, Chellapilla and Fogel [5][6] expanded the game tree after any capture move.

Hence, setting the search depth during learning is a compromise between speed and accuracy, with greater search depths offering better accuracy but slower speed. The current best practice for selecting ply depth during learning seems to be experimental: try several different depths, and see which one results in the best accuracy but slower speed. The current best practice is to use the same ply depth during game play as was used during learning.

III. THE GAME OF GO

The board game Go is usually played on a square grid containing $19 \times 19$ intersections; however, the rules of the game allow it to be played on any size grid. The largest square board which has been solved using computational search alone is a $5 \times 5$ board. Some results based on human analysis in combination with computer-based search exist for $6 \times 6$ and $7 \times 7$ boards [39], [40].

Initially the intersections are empty. The players, Black and White, take turns placing their stones on the empty intersections, but may pass their turn at any time. If both players pass consecutively the game ends. The object of the game is to occupy intersections or enclose them, and the player controlling the most territory or area in the end wins the game. Players can increase their territory by capturing the opponent’s blocks and removing them from the board. A block (also known as a group) is a set of directly connected stones of the same color. A block is captured when the opponent places a stone on its last liberty. A liberty is a directly neighboring empty intersection to a stone within the block. Suicide moves are moves that do not capture an opponent block and produce a block without a liberty. In general suicide moves are illegal. Although these are the main rules of the game there exist several subtle variations. The two main methods of scoring the end game are territory scoring (Japanese rules) and area scoring (Chinese rules). In this work area scoring is used. For a survey of AI approaches to Go see Bouzy and Cazenave [41], and Muller [27].

Go programs have been around since the early 1960’s and over the years a large number of game-specific features have been tried. For example, in [39] a heuristic evaluation function for a small Go board aims to (1) maximize the number of stones on the board, (2) maximize the number of liberties, (3) avoid moves on the edges, (4) connecting stones, and (5) making eyes. The weighted piece counter is only capable of directly representing features (1) and (3).

A board size of $5 \times 5$ is sufficiently complex for the current study. When playing Go on larger boards than this, there is less to be learned about the value of particular positions, since most locations on the board are then defined more by their context in the game rather than possessing any inherent location value. While many Go players believe that Go on anything less than a $9 \times 9$ board is not really Go, smaller boards still offer interesting challenges [40]. Figure 1 shows a $5 \times 5$ board. The left-hand side of the figure shows the 6 unique symmetric positions on the board, while the right side of the board shows the value and color of opening moves at each position, as solved for by van der Werf [40]. These values are useful to compare against the values found by our weighted piece counters.

Also, it may be that the results have some relevance to larger boards, given that many methods address large-board Go by a sliding window, convolutional, or ‘roving eye’ method, where small windows are scanned over the larger board [12] [23]. Note, however, that consideration needs to be given to edge effects, which limit the degree of translation invariance of board features.

![Go board values](image)

Fig. 1. On the left the unique symmetric positions of $5 \times 5$ Go board labelled 1 to 6. On the right the value of opening moves by color and score.

Having a small board, together with a simple weighted piece counter, also makes it easier to compare the different game strategies acquired by the different learning algorithms.

IV. EXPERIMENTAL SETUP

This section describes the aspects of the experimental setup that are common to both the TDL and CEL experiments.
To keep the comparison as fair as possible, each of the experimental runs comparing the relative performance of the two approaches involved the playing of exactly the same number of games: either 45,000, or 435,000, depending on the experiment. The TDL method involves more computation per game, since the weights are updated during play, but this did not measurably add to the time taken per game played.

GNUGo 3.4 was used as the game engine. This generates the set of legal moves for each game state, determines when a game is over, and the final outcome of each game. An upper number of games: either 45,000, or 435,000, depending on the experiment. The TDL method involves more computation per game, since the weights are updated during play, but this did not measurably add to the time taken per game played.

A. The Game State Evaluation Function

The input to the evaluation function is a direct description of the state \( s \), of the board. That is, for an \( m \times m \) board the features, \( \phi_1(s), \ldots, \phi_{m \times m}(s) \), represent the intersection on the board, using \(+1\) for Black, \(-1\) for White, and 0 for empty. The Black player minimizes the evaluation function while White maximizes it.

The simplest evaluation function is the weighted linear function:

\[
h(s, w) = w_0 + w_1 \phi_1(s) + w_2 \phi_2(s) + \ldots + w_n \phi_n(s)
\]

(1)

where \( \phi_i(s) \) is the \( i \)th feature and parameter \( w_i \) the corresponding weight. If the features are predetermined the problem is reduced to determining the weights \( w_i \). Furthermore, one may force the output to be in the range \(-1\) to 1,

\[
f(\text{net}) = \tanh(\text{net}) = \frac{2}{1 + \exp(-2\text{net})} - 1
\]

(2)

For a \( 5 \times 5 \) board, this gives a total of 26 weights to learn - one for each board position, plus a bias weight. Note that for CEL, the bias is not strictly necessary for a single node network in this application. Moves are selected on the relative ordering of their effects on the output, and changes in the bias term cannot affect this (leaving aside floating-point round-off errors). For the CEL, the bias was included as a default part of our neural network implementation.

For TDL, however, the bias does have an effect in two ways. Firstly, TDL learns a utility function: it tries to predict the probability of winning from each position. Changing the bias would change this estimate, although it would not change the moves made for a given set of board position weights. Secondly, changing the bias will affect the derivative of the \( \tanh \) function, which is a term in the weight update equation.

B. Benchmarking Player Performance

The chosen benchmarking method gives an effective assessment of playing ability for players that may differ widely in quality. Although handicapping was considered, this was not used, in order to make the methodology transferable to other games that did not allow for a sensible handicap system. GNUGo 3.4 (level 10) is a strong player on a \( 5 \times 5 \) board, and only the very best CEL or TDL trained players would win any games at all. The approach taken was therefore to modify GNUGo by making random moves with a certain probability \( P_r \). The strength (\( P_r \)) can be varied from purely random to GNUGo at full strength. This probability was fixed to 0.5, and provided an effective measure for the range of players involved in this study. During testing exploration is switched off, i.e. \( \epsilon = 0 \).

C. Player Symmetry

It is convenient to be able to interpret the game state value function from the point of view of either Black or White. Schraudolph et al. [23] used anti-symmetric (\( \tanh \)) squashing functions, then reversed the sign of each piece on the board (using a \((+1, 0, -1)\) piece coding scheme), and also the sign of the bias to achieve this. Given our simpler network the same effect can be achieved by simply changing the sign of the output, since the overall network function of our weighted piece-counter is anti-symmetric.

D. Running the Experiments

Running 10,000 games requires on average 2 minutes, both for TDL and for CEL on a Pentium 4 2.8 GHz, single CPU. The time taken by GNUGo to generate the set of legal moves, and determining the winner once the game is over, far outweighs the computational costs of TDL and CEL. The testing against random GNUGo took around 15 minutes per 10,000 games. Note that RAND/GNUGo is a slower player than the TDL and CEL players. All experiments were performed on a 130-node linux cluster, using various parameter settings, and in total have used thousands of hours of CPU time. Only the most significant of these results are presented.

V. TEMPORAL DIFFERENCE LEARNING (TDL)

In TDL the weights of the evaluation function are updated during game play using a gradient-descent method. Let \( s \) be the after-state observed by a player and similarly \( s' \) the after-state for the player’s next move. Then the evaluation function may be updated during play as follows [19, p.199]:

\[
w_i \leftarrow w_i + \alpha [r + f(\text{net}') - f(\text{net})] \frac{\partial f(\text{net})}{\partial w_i}
\]

(3)

where \( w_i = h(s', \omega) \) and \( \text{net} = h(s, \omega) \). This method is known as gradient-descent TD(0) [19]. If \( s' \) is a terminal state then \( f(\text{net}') \) is interpreted as having the value 0 and the rewards \( r \) correspond to the final utilities, \(+1\) if the winner is Black, \(-1\) when White, and 0 for a draw. The update rule is perhaps the simplest version of temporal difference learning and works quite well on this task. If the step size parameter \( \alpha \), in (3), is reduced properly over time this method will also converge [19, p. 13].

A number of games are played, with each turn made by selecting with highest probability the move resulting in the highest expected utility using a 1-ply minimax search. However, with \( \epsilon \) probability a legal move is selected randomly. These exploratory moves cause the players to experience states
that they might otherwise not experience. In this manner the space of policies is searched. One TD approach to learning an “optimal” or greedy policy is to update only when greedy, non-exploratory, moves are performed. If the value function is updated only when a greedy move is performed, what is learned is the expected return from each position when playing optimally from then on. But this is actually not the case since exploratory moves are made occasionally.

There are a number of parameters that need to be set when using TDL. First of all the algorithm implemented is a
exploratory moves are made occasionally.
learned is the expected return from each position when playi ng
is updated only when a greedy move is performed, what is
non-exploratory, moves are performed. If the value functio n
an “optimal” or
space of policies is searched. One TD approach to learning
that they might otherwise not experience. In this manner the

VI. CO-EVOLUTIONARY LEARNING (CEL)

The closest equivalent to a self-play game in evolutionary learning is co-evolution. In co-evolution individuals within a generation compete with each other. The simplest of all evolutionary algorithms is the (1 + 1) ES. This is essentially a randomized hill-climber and was implemented successfully in [4] for the game of Backgammon. The level of this success is debated in [10] and [9]. Backgammon is a noisy game and so the game itself produces variations in play. This is not the case for a deterministic game like Go, where it is necessary to engineer the variation in play, as discussed in section II-E. These experiments investigate the effects of both population size, and of playing using a noisy policy, on the success of CEL.

In [5], a population-based approach was successfully imple-mented using a simple (15 + 15) ES. Each of the 15 parents creates one offspring. Each of the resulting 30 players then played, using the same color, a randomly selected player

(4)

Fig. 2. Average performance of 30 independent experiments of TDL strategies playing 100 games in each color (i.e. as black and as white) against RAND/GNUGo.

(5)

where the parameters $\kappa$ and $\beta$ must be tuned for each problem [43], [44].

This leads us to the co-evolutionary approach applied here. The approach uses a single parent, like [4], but a larger number of offspring are produced. This is the $(1 + \lambda)$ ES, where $(1, \lambda)$ denotes that the parent does not compete with the offspring and $(1 + \lambda)$ ES denotes that it does. At each generation an individual is matched up against every other player in the population twice, each time playing a different color. A score of $+1$ is given for a win, $-2$ for a loss, and $0$ for a draw. A total of $\lambda(\lambda - 1)$ games are therefore played per generation. The complete algorithm is presented in figure 3. For a fixed mutation strength lines 4 and 10 in figure 3 are deleted and $\sigma = \sigma'$. The algorithm uses a non-isotropic mutative self-adaptive scheme, that is, each weight has its own mutation strength lines

(5)

where the parameters $\kappa$ and $\beta$ must be tuned for each problem [43], [44].

This leads us to the co-evolutionary approach applied here. The approach uses a single parent, like [4], but a larger number of offspring are produced. This is the $(1 + \lambda)$ ES, where $(1, \lambda)$ denotes that the parent does not compete with the offspring and $(1 + \lambda)$ ES denotes that it does. At each generation an individual is matched up against every other player in the population twice, each time playing a different color. A score of $+1$ is given for a win, $-2$ for a loss, and $0$ for a draw. A total of $\lambda(\lambda - 1)$ games are therefore played per generation. The complete algorithm is presented in figure 3. For a fixed mutation strength lines 4 and 10 in figure 3 are deleted and $\sigma = \sigma'$. The algorithm uses a non-isotropic mutative self-adaptive scheme, that is, each weight has its own mutation strength associated with it. The algorithm is initialized with $\sigma = 0$, as in the TDL experiments, and $\sigma = 1/\sqrt{n}$ which is also its upper bound. There are a number of parameter settings

(5)

where the parameters $\kappa$ and $\beta$ must be tuned for each problem [43], [44].

This leads us to the co-evolutionary approach applied here. The approach uses a single parent, like [4], but a larger number of offspring are produced. This is the $(1 + \lambda)$ ES, where $(1, \lambda)$ denotes that the parent does not compete with the offspring and $(1 + \lambda)$ ES denotes that it does. At each generation an individual is matched up against every other player in the population twice, each time playing a different color. A score of $+1$ is given for a win, $-2$ for a loss, and $0$ for a draw. A total of $\lambda(\lambda - 1)$ games are therefore played per generation. The complete algorithm is presented in figure 3. For a fixed mutation strength lines 4 and 10 in figure 3 are deleted and $\sigma = \sigma'$. The algorithm uses a non-isotropic mutative self-adaptive scheme, that is, each weight has its own mutation strength associated with it. The algorithm is initialized with $\sigma = 0$, as in the TDL experiments, and $\sigma = 1/\sqrt{n}$ which is also its upper bound. There are a number of parameter settings

(5)

where the parameters $\kappa$ and $\beta$ must be tuned for each problem [43], [44].

This leads us to the co-evolutionary approach applied here. The approach uses a single parent, like [4], but a larger number of offspring are produced. This is the $(1 + \lambda)$ ES, where $(1, \lambda)$ denotes that the parent does not compete with the offspring and $(1 + \lambda)$ ES denotes that it does. At each generation an individual is matched up against every other player in the population twice, each time playing a different color. A score of $+1$ is given for a win, $-2$ for a loss, and $0$ for a draw. A total of $\lambda(\lambda - 1)$ games are therefore played per generation. The complete algorithm is presented in figure 3. For a fixed mutation strength lines 4 and 10 in figure 3 are deleted and $\sigma = \sigma'$. The algorithm uses a non-isotropic mutative self-adaptive scheme, that is, each weight has its own mutation strength associated with it. The algorithm is initialized with $\sigma = 0$, as in the TDL experiments, and $\sigma = 1/\sqrt{n}$ which is also its upper bound. There are a number of parameter settings

(5)

where the parameters $\kappa$ and $\beta$ must be tuned for each problem [43], [44].

This leads us to the co-evolutionary approach applied here. The approach uses a single parent, like [4], but a larger number of offspring are produced. This is the $(1 + \lambda)$ ES, where $(1, \lambda)$ denotes that the parent does not compete with the offspring and $(1 + \lambda)$ ES denotes that it does. At each generation an individual is matched up against every other player in the population twice, each time playing a different color. A score of $+1$ is given for a win, $-2$ for a loss, and $0$ for a draw. A total of $\lambda(\lambda - 1)$ games are therefore played per generation. The complete algorithm is presented in figure 3. For a fixed mutation strength lines 4 and 10 in figure 3 are deleted and $\sigma = \sigma'$. The algorithm uses a non-isotropic mutative self-adaptive scheme, that is, each weight has its own mutation strength associated with it. The algorithm is initialized with $\sigma = 0$, as in the TDL experiments, and $\sigma = 1/\sqrt{n}$ which is also its upper bound. There are a number of parameter settings

(5)

where the parameters $\kappa$ and $\beta$ must be tuned for each problem [43], [44].

This leads us to the co-evolutionary approach applied here. The approach uses a single parent, like [4], but a larger number of offspring are produced. This is the $(1 + \lambda)$ ES, where $(1, \lambda)$ denotes that the parent does not compete with the offspring and $(1 + \lambda)$ ES denotes that it does. At each generation an individual is matched up against every other player in the population twice, each time playing a different color. A score of $+1$ is given for a win, $-2$ for a loss, and $0$ for a draw. A total of $\lambda(\lambda - 1)$ games are therefore played per generation. The complete algorithm is presented in figure 3. For a fixed mutation strength lines 4 and 10 in figure 3 are deleted and $\sigma = \sigma'$. The algorithm uses a non-isotropic mutative self-adaptive scheme, that is, each weight has its own mutation strength associated with it. The algorithm is initialized with $\sigma = 0$, as in the TDL experiments, and $\sigma = 1/\sqrt{n}$ which is also its upper bound. There are a number of parameter settings

(5)

where the parameters $\kappa$ and $\beta$ must be tuned for each problem [43], [44].

This leads us to the co-evolutionary approach applied here. The approach uses a single parent, like [4], but a larger number of offspring are produced. This is the $(1 + \lambda)$ ES, where $(1, \lambda)$ denotes that the parent does not compete with the offspring and $(1 + \lambda)$ ES denotes that it does. At each generation an individual is matched up against every other player in the population twice, each time playing a different color. A score of $+1$ is given for a win, $-2$ for a loss, and $0$ for a draw. A total of $\lambda(\lambda - 1)$ games are therefore played per generation. The complete algorithm is presented in figure 3. For a fixed mutation strength lines 4 and 10 in figure 3 are deleted and $\sigma = \sigma'$. The algorithm uses a non-isotropic mutative self-adaptive scheme, that is, each weight has its own mutation strength associated with it. The algorithm is initialized with $\sigma = 0$, as in the TDL experiments, and $\sigma = 1/\sqrt{n}$ which is also its upper bound. There are a number of parameter settings

(5)

where the parameters $\kappa$ and $\beta$ must be tuned for each problem [43], [44].

This leads us to the co-evolutionary approach applied here. The approach uses a single parent, like [4], but a larger number of offspring are produced. This is the $(1 + \lambda)$ ES, where $(1, \lambda)$ denotes that the parent does not compete with the offspring and $(1 + \lambda)$ ES denotes that it does. At each generation an individual is matched up against every other player in the population twice, each time playing a different color. A score of $+1$ is given for a win, $-2$ for a loss, and $0$ for a draw. A total of $\lambda(\lambda - 1)$ games are therefore played per generation. The complete algorithm is presented in figure 3. For a fixed mutation strength lines 4 and 10 in figure 3 are deleted and $\sigma = \sigma'$. The algorithm uses a non-isotropic mutative self-adaptive scheme, that is, each weight has its own mutation strength associated with it. The algorithm is initialized with $\sigma = 0$, as in the TDL experiments, and $\sigma = 1/\sqrt{n}$ which is also its upper bound. There are a number of parameter settings

(5)
that need to be investigated for this algorithm and they form the basis for the experimental results presented in this section. They are:

- elitist $(1 + \lambda)$ versus non-elitist $(1, \lambda)$,
- parameters $\beta$ and $\kappa$,
- self-adaptive versus fixed $\sigma$,
- population size $\lambda$,
- and finally the introduction of noise to the policy.

A number of different parameter settings were tried. The experimental results presented here are representatives of these experiments.

The first comparison concerned the elitist $(1 + 9)$ ES and non-elitist $(1, 10)$ ES using a fixed mutation rate $\sigma = 1/\sqrt{n}$. Two variations are tried, one where $\beta = 0.05$, as proposed in [4], and another where $\beta = 1$ (no average). The average result of 30 independent experiments, each playing RAND/GNUGo at every generation for benchmarking purposes only, is depicted in figure 4. Similar behavior is observed for larger population sizes and when using self-adaptation. The results show that averaging using $\beta = 0.05$ is useful for both elitist and non-elitist runs, and that the elitist ES is slower to learn than its non-elitist counterpart. The level of play is, nevertheless, poorer than the one achieved using TDL.

The next comparison involved the self-adaptive versus fixed-strength mutation operators. Only the non-elitist strategies $(1, \lambda)$ are considered. Additionally, experiments using $\lambda = 30$ are added. These experimental results, for $(1, 10)$ and $(1, 30)$ ES using $\beta = 0.05$, are presented in figure 5. In the case of self-adaptation $\kappa = 0.2$. The main results from these runs are that for smaller populations $\lambda$ using a fixed mutation rate may be more successful but for larger populations the difference is indistinguishable. The most interesting result, however, is that a higher level of play is achieved by increasing the population size.

The level of play has reached that of TDL but at an increased cost. Each run for the $(1, 30)$ ES is now based on $(30 \times 29 \times 500 = 435,000)$ games, compared with only 45,000 for TDL.

The previous experiment raises the question of why the larger population improves the ultimate level of play. There are two possible answers:

- The greater number of games played leads to a more statistically reliable measure of the quality of play.
- The greater variation in play that results from a larger population leads to more exploration of policy space.

This led us to the following experiment. The $(1, 10)$ and $(1, 30)$ experiments using self-adaptation, $\beta = 0.05$ and $\kappa = 0.2$, are compared with an equivalent experimental setup using an $\epsilon$-greedy policy, that is with $\epsilon = 0.1$ probability a purely random legal move is performed. This result is shown in figure 6 and clearly illustrate that better players are evolved when noise is added in this manner. Additional variation in play is therefore useful but so too is the increased number of games. The $(1, 10)$ with added noise still does not reach the level of play of TDL, but the $(1, 30)$ ES now surpasses it, as confirmed by the statistical tests in table I.

One of the critical factors is $\beta$. The parameter $\kappa$ is not as critical. When $\beta$ is increased from 0.05 to 0.2 the learning is initially faster but flattens out to a lower level of play. This level of play is still superior to the case when $\beta = 0.05$. The greatest number of games played leads to a more statistically reliable measure of the quality of play.

1. Initialize: $\sigma' := 1/\sqrt{n}$, $w' := 0$, $\tau' = 1/\sqrt{2n}$, $\tau = 1/\sqrt{2\sqrt{n}}$
2. while termination criteria not satisfied do
3. for $k := 1$ to $\lambda$ do (replication)
4. $\sigma_{k,j} := \sigma_j' \exp(\tau' N(0, 1) + \tau N_j(0, 1))$, $j = 1, \ldots, n$
5. $w_k := w' + \sigma_{k,j} N(0, 1)$
6. od
7. if elitist $(1 + (\lambda - 1))$ strategy do
8. $(w_1, \sigma_1) := (w', \sigma')$ (first offspring is replicated unmodified)
9. od
10. each individual $w_k$, $k = 1, \ldots, \lambda$ plays another (once each color), for a total of $\lambda(\lambda - 1)$ games,
11. find the player $i$ with the highest score (breaking ties randomly)
12. $w' := w' + \beta(w_i - w')$ (arithmetic average)
13. $\sigma' := \sigma'/(\sigma_i/\sigma')^\alpha$ (geometric average)

120
140
160
180
200
220
240
260
280
300
320
340
360
380
400
420
440
460
480
500
mean number of games won against RAND/GNUGo
20
40
60
80
100
120
140
160
180
200
220
240
260
280
300
320
340
360
380
400
420
440
460
480
500
mean number of games won against RAND/GNUGo

Fig. 3. The $(1 + \lambda)$ evolution strategy. For each win the player receives a score of 1, 0 for a draw, and $-2$ for loss.
VII. DISCUSSION OF EXPERIMENTAL RESULTS

The strategies learned by both TDL and CEL are limited by what a weighted piece counter can learn. Using a 1-ply minimax search the players aim to maximize their gain, where the gain is the weight corresponding to the intersection played and the sum of the weights corresponding to the area captured. Given the values of the learned weights, it can be seen that both TDL and CEL learn to occupy as much territory as possible, and to avoid the edges. Beyond this, they differ in some of the detailed ordering of the board positions, and in the tradeoffs involved between playing a desirable position versus making a capture.

Consider the following example. In figure 8 typical weights learned using TDL are given. These weights are for a greedy algorithm described by [5] and \( \beta = 1 \) all-versus-all. Black follows with a move at this intersection (move 15) and at move 16 White fills in the eye (where stone 9 was captured). Clearly it is difficult for a weighted piece counter to learn the concept of an eye.

It is possible to determine from the weight values learned which regions of the board are most important. Using the labelling in the left of figure 1 it can be seen that two of the opposing position 5’s are most important, followed by the remaining center positions 4, 5, and 6. The less favorable positions are the edges, of which position 2 is best followed by 3, and the worst is position 1 (the corners). The weight values have been rounded off to six decimal places and so there exist two weights of equal value. In the case of equivalent gain, ties are broken randomly, resulting in two different games played. These games are illustrated in figure 8, where there is either a decisive win for Black or for White. The two games are essentially different at moves 5 and 6 (both having the same gain). It is interesting to plot the utility function for these two games, see figure 9. Here two interesting observations are made. Firstly, different predicted end game utilities are first observed at move 20 and so a deep minimax search is required to make the correct decision at move 5. Secondly, initially the utility function is in favor of White winning, however, on further examination it is found that Black wins just as frequently as White when \( \epsilon \) noise is added to the policy. It appears that the weighted piece counter cannot predict accurately the expected end game utility for both initial and end games states.

It is interesting to see if similar weight values are found when TDL plays against an opponent, say GNUGo at full strength. These weight values are depicted in figure 10. The weights are different. The center position 6 is now the most...
is different from that of self-play TDL. When these two players are matched a number of different games are produced where the self-play strategy wins 83% of the time.

$$-0.051914 \quad -0.051921 \quad -0.051921 \quad -0.051922 \quad -0.051911$$

$$-0.051922 \quad -0.167496 \quad -0.167496 \quad -0.167496 \quad -0.051920$$

$$-0.051922 \quad -0.167496 \quad -0.167496 \quad -0.167496 \quad -0.051915$$

$$-0.051912 \quad -0.051922 \quad -0.051919 \quad -0.051921 \quad -0.051911$$

Fig. 10. The learned weight using TD(0) when playing against GNUGo. The bias value is $$w_0 = -0.015331$$.

We now examine the strategy learned by CEL, in particular a typical player from the noisy (1, 30) ES runs in figure 6. Typical weight values are given in figure 11. In this case the most important intersections are position 5, followed by the 4 and 6. The order of importance for the edges is the same as for the self-play TDL.

$$0.042168 \quad -0.051701 \quad -0.015329 \quad -0.043325 \quad 0.025249$$

$$-0.077448 \quad -0.102016 \quad -0.184831 \quad -0.145874 \quad -0.054164$$

$$0.011327 \quad -0.217861 \quad -0.130290 \quad -0.192060 \quad -0.007380$$

$$-0.054917 \quad -0.138783 \quad -0.169954 \quad -0.132444 \quad -0.070210$$

$$0.069588 \quad -0.086541 \quad -0.011339 \quad -0.029677 \quad 0.072525$$

Fig. 11. Typical CEL weights found using the (1, 30) ES, $$\epsilon$$-greedy. The bias value is $$w_0 = 0.002587$$.

Is the CEL performance in figure 6, in particular the (1, 30) ES using the $$\epsilon$$ noisy policy, statistically different from the TDLs in figure 2? To answer this question, a statistical test of the performance for the second half of the runs for all 30 independent experiments was performed. That is, a comparison of 30 × 250 games against RAND/GNUGo for both learning strategies was made. The medians for the TDLs and CELs are statistically significantly different ($$P < 0.01$$) using the Wilcoxon rank sum test. The median, mean, standard deviation, and standard error of all learners are given in table 1. From the table one may see that the median number of wins is highest for the (1, 30) ES using an $$\epsilon$$ greedy policy, followed by the TDLs, and lastly the noise-free (1, 30) ES. The ratio of White wins to Black wins is 0.92 for both TDL and CEL.

The question arises as to whether CEL only beat TDL due to each evolutionary run being based on ten times as many games. From the graph in figure 2 it appeared that TDL had converged well within the 45,000 games that each initial run was based on, but to test this more rigourously, all the TDL experiments were repeated using 435,000 games (the same as CEL). For all the TDL variants, the near 10-fold increase in the number of games brought no significant improvement, and CEL retained its slight but significant advantage.

One final comparison was made using the final 30 TDL and CEL players, trained with the best configuration of each method from table I i.e., TDL – c.) and CEL (1, 30) ES $$\epsilon$$-greedy. Each of the 60 players were matched with every other, and played in a league both as black and as white. In the top 10 positions there were 5 CEL players and 5 TDL players, indicating that the TDL and CEL achieved very similar levels of play when measured in this way.
TABLE I

<table>
<thead>
<tr>
<th></th>
<th>median</th>
<th>mean</th>
<th>st. dev</th>
<th>st. err</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDL – a.) (45000 games)</td>
<td>157</td>
<td>156.6</td>
<td>7.6</td>
<td>0.09</td>
</tr>
<tr>
<td>TDL – b.)</td>
<td>157</td>
<td>156.7</td>
<td>7.3</td>
<td>0.08</td>
</tr>
<tr>
<td>TDL – c.)</td>
<td>159</td>
<td>158.3</td>
<td>6.7</td>
<td>0.06</td>
</tr>
<tr>
<td>TDL – a.) (450000 games)</td>
<td>157</td>
<td>156.4</td>
<td>7.6</td>
<td>0.09</td>
</tr>
<tr>
<td>TDL – b.)</td>
<td>157</td>
<td>156.8</td>
<td>7.4</td>
<td>0.09</td>
</tr>
<tr>
<td>TDL – c.)</td>
<td>158</td>
<td>158.0</td>
<td>6.7</td>
<td>0.08</td>
</tr>
<tr>
<td>CEL (1, 30) ES e-greedy</td>
<td>161</td>
<td>159.8</td>
<td>8.6</td>
<td>0.10</td>
</tr>
<tr>
<td>CEL (1, 30) ES</td>
<td>155</td>
<td>154.7</td>
<td>8.6</td>
<td>0.10</td>
</tr>
</tbody>
</table>

VIII. CONCLUSION

This paper reported a thorough investigation of temporal difference versus evolutionary learning to adapt the parameters of a position weighted piece counter for small-board Go. For each method (TDL and CEL), the search space, defined by the set of parameters of the weighted piece counter, was identical. Only the method for adjusting the parameters was different. Each method bootstrapped itself to achieve a much improved level of play over its initial configuration. That is, each method improved its play without the need to play against a human-designed player. The main result is that TDL learns much faster, but that CEL eventually achieves a higher standard of play.

There is a good reason why TDL might be expected to learn faster than CEL, since the TDL method uses more information from each game to assign credit to particular parameters of the evaluation function. This makes the search much more directed than the relatively blind evolutionary search, providing that the information available during play is actually useful. If this were not the case, then trying to utilize such information could in principle hinder learning.

Regarding our experiments with CEL, the findings were as follows:

- for the weighted piece counter there is no need for self-adaptation, the weights are free to adjust themselves to a value befitting the mutation strength;
- it is necessary to average between parent and offspring to get best performance, a known technique for dealing with noisy fitness functions [43];
- the elitist strategies exhibit long periods of stagnation and so are slower to learn than the non-elitist strategies;
- adding noise to the policy during evolution, and using larger populations significantly enhances learning;

All of the above steps were necessary in order for CEL to outperform TDL, and even then, it required nearly 10 times as many games per experimental run. Given this, CEL outperformed TDL by a small but statistically significant margin. Increasing the number of games used by TDL did not improve its performance. The reason for this is that the CEL players are exposed to more realistic game play variation than is possible with a single self-play TDL player. This conclusion is supported by the improvement in play observed when increasing the population size from 10 to 30. Although the larger population runs also involved more games being played, all experiments appeared to run to convergence, so playing more games with the smaller sized population would not improve final performance. Investigating CEL/TDL hybrids, such as using a population of TDL players is therefore a promising direction for future work.

Both CEL and TDL can be improved with the aid of human-designed board features and neural network architectures. Another interesting research direction would involve CEL/TDL hybrids, where the evolutionary algorithm was responsible for constructing game-board features, and details of the network architecture, and TDL was responsible for tuning the feature and network parameters. This mix of using evolution for feature selection and architecture specification, and gradient descent for setting the weights, is common when evolving neural network architectures for subsequent supervised training [45], but has not yet been reported in co-evolutionary game function learning.

Note that given an arbitrary feed-forward neural network constructed from differentiable function units (such as sums, products and Tanh functions) it is straightforward to automatically generate the network parameter derivatives needed by TDL. One simple way to achieve this is to replace all the usual feed-forward nodes with forward-backward nodes [46].

There is much more yet to be learned regarding the type of situations in which TDL or CEL perform best. Like backgammon, Go has the feature that much of the board state remains stable: once pieces are played, there is a reasonable chance that they will remain there until the end of the game, at which point the true rewards are known. This is in contrast to a game such as checkers, for example, where the board states are usually much less stable.

All the learning algorithms studied achieved a much improved level of play over the initial player configurations. However, changes to the details of each algorithm can have a very significant impact on the ultimate level of play obtained.

Acknowledgements

The authors thank the anonymous reviewers, and David Fogel, Yngvi Björnsson, and Bruno Bouzy, for their helpful and insightful comments on earlier versions of this paper.

REFERENCES


