Investigating Learning Rates for Evolution and Temporal Difference Learning

Simon M. Lucas, Senior Member IEEE

***Abstract*— Evidently, any learning algorithm can only learn on the basis of the information given to it. This paper presents a first attempt to place an upper bound on the information rates attainable with standard co-evolution and with TDL. The upper bound for TDL is shown to be much higher than for co-evolution. Under commonly used settings for learning to play Othello for example, TDL may have an upper bound that is hundreds or even thousands of times higher than that of co-evolution.**

To test how well these bounds correlate with actual learning rates, a simple two-player game called *Treasure Hunt* is developed. While the upper bounds cannot be used to predict the number of games required to learn the optimal policy, they do correctly predict the rank order of the number of games required by each algorithm.

I. Introduction

One of the key problems in machine learning is how an agent can best learn in a largely unsupervised manner via interactions with its environment. The payoff for a particular action may be far removed in time and in state-space from the action's execution. The foremost methods for addressing this problem are reinforcement learning algorithms such as temporal difference learning (TDL), and evolution (or depending on the problem setup, co-evolution).

TDL was applied by Samuel as far back as 1957 [1] and Michie in 1961 [2]. A famously successful application of TDL was Tesauro's TD Gammon [3], which was followed up by an evolutionary approach to the same problem by Pollack and Blair [4]. In recent years there has been a surge of interest in evolutionary approaches to this type of learning. Much of this was probably inspired by the work of Pollack and Blair [4], and Chellapilla and Fogel [5] [6] [7].

It is relevant for the following sections to consider the population sizes typically used in co-evolutionary learning. Pollack and Blair used a population size of two (a random mutation hill-climber), and used parent-child weighted averaging to overcome effects of noise. Fogel and Chellapilla used a population size of 30 (as a $(15 + 15)$ ES), though using random opponent selection rather than a full round-robin league. Runarsson and Lucas [8] experimented with various population sizes, but found sizes of 10 or 30 to work well (anything between that would work well also).

It is also important to consider the relative performance achieved by TDL and co-evolution. Runarsson and Lucas [8] and Lucas and Runarsson [9] conducted extensive comparisons for learning weighted piece counters for playing small board go, and for Othello. They introduced forced random moves into the game play in order to provide a more robust test for the learning algorithms. They found that TDL typically learned much faster then co-evolution, and that standard co-evolution scarcely learned anything at all. By introducing parent-child weighted averaging, however, and after playing millions of games, co-evolution eventually outperformed TDL. Notably, they only found this for simple weighted piece counters (with only 64 weights to learn for Othello).

However, on the Othello Neural Network Server[1] all the best players are based on more complex architectures with many more weights than 64. The best performing architectures are (in best first order) symmetric n-tuple networks, spatial MLPs, MLPs, weighted piece counters. The best performing symmetric n-tuple networks have around 15,000 weights. Apart from the weighted piece counters, all the best networks on the server have been trained using TDL, though the IEEE CEC 2006 competition winner improved on a TDL-trained MLP by slightly adapting its weights using an evolutionary algorithm.

Co-evolution is attractive due to its simplicity and its robust emphasis on overall performance. Unfortunately, it may be that for large complex architectures which are necessary to represent (and hence learn) complicated value functions leading to interesting behaviour, co-evolution may be too wasteful to do this well in practice. The results from neural network Othello mentioned above support this hypothesis.

Learning game strategy is similar to learning any kind of control strategy, and also relevant are the results of Gomez *et al* [10]. They showed that a type of evolutionary algorithm called CoSyNE (Cooperative Synapse Neuroevolution) significantly outperformed all other methods under test, on difficult pole balancing problems. They also showed that TDL methods were especially bad for that problem. However, the neural networks evolved were of modest size, with around 40 weights. Furthermore, CoSyNE is a non-standard EA that exploits more information from the environment than the standard type of EA being studied in this paper. CoSyNE performs a type of credit assignment whereby synapses are evolved on the basis of how well the networks they contribute to perform. This gleans more information from the control or game environment than simply picking a winner. It would also be interesting to calculate information rates for CoSyNE in a future paper.

The contribution of this paper is to apply some basic

Simon Lucas is with the Department of Computing and Electronic Systems, University of Essex, Colchester CO4 3SQ, UK, email: sml@essex.ac.uk

[1] http://algoval.essex.ac.uk:8080/othello/html/Othello.html

principles of information theory as laid down by Shannon [11] to place upper bounds on how much information can be gained per game played by each of these learning algorithms. It will be shown that the upper bound for TDL is always higher than for single-parent co-evolution, and usually very much higher. A simple game called the *Treasure Hunt Game* is then described. This game was designed specifically to analyse how well these learning rates compare with the rates achieved in practice for a simple game. Note that for more complex games the actual rates are likely to be lower.

Experiments were performed to measure the average number of games needed in order to learn the optimal policy for this game. TDL is shown to learn the optimal policy much faster than co-evolutionary learning, but the rate of information acquisition in each case is shown to be significantly lower than the upper bound.

The rest of this paper is structured as follows. Section II calculates maximum possible learning rates for single-parent selection evolutionary algorithms, and for temporal difference learning. Section III describes the Treasure Hunt Game. Section IV reports empirical results for learning the optimal value function for that game, and section V concludes.

II. Information Rates

In this section we calculate upper bounds on the information rates attainable with each algorithm, in terms of the average number of bits of information per game (bg^{-1}). The analysis is developed first in general terms, and then figures are chosen from recent experimental work on game strategy learning in order to gain an idea of a typical ratio of maximum information rates between the two approaches.

It is assumed that each learning algorithm operates by learning a vector of parameters (weights) that control the behaviour of a player in some way. The exact nature of the mapping from the parameters to the behaviour induced by those parameters makes no difference to the calculation of the information rate upper bound, though in practice the mapping has a profound effect on the actual learning rate, and on the level of play that the learner eventually achieves.

A standard example is learning the weights of a multi-layer perceptron (MLP), where the MLP is used as a value function within a game-playing engine. The analysis is applicable to most types of game, but here to give a concrete example we choose the two-player board game Othello.

Although it is stated elsewhere in the paper, it is worth emphasising that the information rates calculated below are *upper bounds*. This rests on the assumption that all events under consideration are equally likely. In the case of evolutionary algorithms, all members of the next generation are assumed equally likely to be the best. In the case of TDL, only the branching factor of the game is considered and the assumption is that all legal moves are equally likely.

A. Evolution

The analysis developed here can be applied equally well to evolutionary or co-evolutionary algorithms. The analysis is done for a $(1 \stackrel{+}{,} \lambda)$ Evolution Strategy (ES) - these are commonly used and have been found to perform competitively with other evolutionary algorithms for game strategy learning [8] [9]. This choice of EA makes the calculation straightforward. In a thorough comparison of various co-evolution settings, Runarsson and Lucas [8] found that best performance was obtained when selecting a single parent from each generation, but then using weighted averaging of the parent and best child to form the next parent; this was necessary to counteract the effects of noise that resulted from forced random play.

Note that it is important to distinguish between information and data. For example, when initialising an MLP with random weights, we are transferring lots of data into the network, but no information. Similarly, when a child network inherits most of the weights from a parent, lots of data may be transferred, but this is not the same as the information gained. Information is only gained in the evaluation and selection process. It is also possible to use evolutionary algorithms with case injection [12] [13], but in that case the information is being given to the evolutionary algorithm in the form of a previously acquired case-base. The analysis developed in this paper would still apply to the information gained as a result of running the evolutionary algorithm.

Suppose that each generation of players (where each player p_i is controlled by a weight vector w_i) is evaluated by playing every other player in a round-robin league. Since for our choice of ES there are $n = \lambda$ or $n = (\lambda + 1)$ players, this leads to $n(n-1)$ games being played per generation (for the two player board game example, each player plays every other player once as black and once as white). Next we calculate the maximum information gained by the learner each generation, where the learner in this case is a single-parent evolutionary algorithm.

At each generation a single parent is chosen from a possible n parents. Usually, the player that finishes top of the league for that generation is chosen. Other selection policies are also possible such as a $(\lambda + \lambda)$ ES (hence population size of $n = 2\lambda$). However, while selecting more parents (for a given population size) might seem to increase the information used by the algorithm, there are two points to note regarding this. Firstly, as less able parents are selected, so the usefulness of the information decreases. In terms of information rates, we would be using the same amount of information if we always picked the worst player as a parent, but the utility of that information would now be negative, assuming that the intention was for good play to evolve. Secondly, as more individuals are selected, the added information declines (irrespective of the utility of the information). The maximum information is reached when half the population is selected. After that selecting more decreases the amount of information, since it would be more efficient to specify the complement of the selected set. In other words, the information can be encoded in fewer bits by specifying the set of non-parents rather than the set of parents. If an evolutionary algorithm was able to make full use of the entire ranking of the population, then this could

n	$I_c(\text{bg}^{-1})$
2	0.500
5	0.12
10	0.037
30	0.006

TABLE I

UPPER BOUNDS FOR INFORMATION GAINED DURING GAME PLAY FOR CO-EVOLUTION WHEN USING A ROUND-ROBIN LEAGUE (UNITS OF BITS PER GAME).

be directly specified in $n\log_2 n$ bits, simply by tagging each of the n members of the population with their position in the ranked list.

To encode this information (i.e. the choice of 1 parent from a set of n possible ones) requires at most $\log_2 n$ bits. Therefore the information rate in units of bits per game for single-parent selection co-evolution I_c is given by:

$$I_c = \frac{\log_2 n}{n(n-1)} \tag{1}$$

This is an upper bound because it assumes that each member of the population is equally likely to be the fittest. In many cases this assumption is invalid. For example, consider a simple single-bit mutation evolutionary algorithm for solving the one-max problem (a bit string problem where the fitness is defined as the number of bits set to one). If the bit to be mutated is chosen uniformly randomly over the entire string, then as the algorithm gets closer to the optimum, the probability of flipping a zero to a one decreases. Hence, most randomly mutated children of the parent will be worse than the parent. This discussion can be directly applied to the Treasure Hunt Game described below, using a bit-vector to specify the policy. This becomes similar to the one-max problem in terms of the solution space, though the fitness landscape is very different, since game fitness is determined with respect to the opponent or set of opponents.

Table I shows how the upper bound varies with the population size for some commonly used population sizes.

An alternative to using co-evolution is to use standard evolution where the fitness function is based on performance against a fixed player, or a fixed set of players. Assume now that each member of the population is evaluated by playing m games against the set of fixed players, the information rate upper bound for standard evolution I_e is

$$I_e = \frac{\log_2 n}{mn} \tag{2}$$

Another alternative is to use knock-out tournament selection within a co-evolutionary algorithm. There are many ways this can be applied, and most will lead to higher information rate upper bounds than a round-robin league. However, tournament selection is less reliable than a round-robin league, since the tournament winner may not be the best player (unless the game is transitive and deterministic, in which case the winner of a knock-out tournament *is* guaranteed to be the best player). A knock-out tournament requires $2(n-1)$ games to select a winner from n players (the factor of 2 comes from the board-game context where each player plays as black then as white), leading to the information rate in equation 3.

$$I_e = \frac{\log_2 n}{2(n-1)} \tag{3}$$

Tournament selection can also be used with smaller tournament sizes, and in each case the information rate upper bound calculation is straightforward.

Evolutionary algorithms make progress when the selected child is fitter than the parent. As the algorithm converges toward an optima a reasonable assumption is that the probability of an undirected random mutation leading to an improvement decreases, thereby lowering entropy and hence the information content. Under this assumption, actual information acquisition rates will fall short of these upper bounds. Whether this assumption holds depends on the nature of the fitness landscape. Also, adaptive mutation rates or more sophisticated evolutionary algorithms such as CMA may alleviate this effect by making directed mutations.

B. Temporal Difference Learning

Temporal difference learning extracts unsupervised information from the game as it is played, and also uses supervised reward signals when available, in this case at the end of each game. The purpose of the unsupervised learning phase is to acquire a model of how states are temporally related i.e. which game states can be reached from which other game states.

In TDL the weights of the evaluation function are updated during game play using a gradient-descent method. Let $\vec{x}$ be the board observed by a player about to move, and similarly $\vec{x}'$ the board after the player has moved. Then the evaluation function may be updated during play as follows.

$$\begin{aligned} w_i &\leftarrow w_i + \alpha\big[v(\vec{x}') - v(\vec{x})\big]\frac{\partial v(\vec{x})}{\partial w_i} \\ &= w_i + \alpha\big[v(\vec{x}') - v(\vec{x})\big]\big(1 - v(\vec{x})^2\big)x_i \end{aligned} \tag{4}$$

where

$$v(\vec{x}) = tanh(f(\vec{x})) = \frac{2}{1+\exp(-2f(\vec{x}))} - 1 \tag{5}$$

is used to force the value function v to be in the range -1 to 1. This method is known as gradient-descent TD(0) [14]. If $\vec{x}'$ is a terminal state then the game has ended and the following update is used:

$$w_i \leftarrow w_i + \alpha\big[r - v(\vec{x})\big]\big(1 - v(\vec{x})^2\big)x_i$$

where r corresponds to the final utilities: $+1$ if the winner is Black, -1 when White, and 0 for a draw.

This is based on Sutton and Barto [14, p.199], and the formulation of it in Equation 4 together with the following explanation is taken directly from Lucas and Runarsson [9].

At each turn of the game, the TDL player either makes an in-game or a terminal (end-game) update. In the case of

an in-game update, the value of the previous board position is adjusted to be more similar to the value of the current board position. This is a type of bootstrapping process. For a terminal update, the value of the penultimate board is adjusted to be closer to the final value of that game ($r = +1$ for black win, $r = 0$ for draw, $r = -1$ for white win).

Interfacing a TDL-learner to a game engine is straightforward. The game engine calls a TDL update method for any TDL player after each move has been made: it calls inGameUpdate during a game, or terminalUpdate at the end of a game.

It is instructive to study the Java code that implements this process as shown in Figure 1. The variables are as follows: `op` is the output of the network; `tg` is the target value; `alpha` is the learning rate; `delta` is the back error term; `prev` is the previous state of the board; `next` is the current state of the board; `net` is an instance variable bound to some neural network type of architecture.

In the case of Othello there are a maximum 3^{64} possible game states (since each square can be in at most 3 states, empty, black, or white). If each square is equally likely to be in any of the three states, this would lead to $\log_2 3^{64} \approx 101$bits of information being available at each move. [2]

However, successive states are highly correlated, so any measure based directly on this would provide an overly loose upper bound.

Arguably a better way to proceed is to consider the branching factor of the game. If a game has on average b_r possible moves at each of s stages, then each move conveys on average $\log_2 b_r$ bits of information, hence during the game we get $(s-1)$ times this information, because the last update is based on the true reward. The true reward could be win, draw, or lose, leading to an upper bound of $\log_2 3$ bits of information available at the end of the game.

Hence, the information available I_t for a TDL learner during a game is given by:

$$I_t = (s-1)\log_2 b + \log_2 3 \tag{6}$$

noting that $s \geq 1$ and $b_r > 1$. From equations 1 and 6 it is clear that the upper bound for TDL is always greater than for single parent co-evolution.

Putting the appropriate figures in for Othello for $b_r = 7$ and $s = 60$ leads to the figure of $I_t \approx 166bg^{-1}$.

For the Treasure Hunt Game we can use a better estimate than the average branching factor, since at each move we know exactly the number of possible moves. If they are assumed equally likely (not true of course, since players develop hypotheses about which squares contain treasure as the set of games are played), then the upper bound can be calculated as follows:

$$I_t = \log_2 3 + \sum_{i=1}^{n} \log_2 i \tag{7}$$

[2]The true figure is lower than this, since the centre four squares can never be empty, but the only purpose of stating this is to give a rough estimate, and this figure will not be used in the subsequent calculations.

which is approximately 298 bits in the case of game size $n = 64$.

Considering the way in which temporal difference learning is usually applied to game strategy learning i.e. to train a function approximator, it should be apparent that this is a very loose upper bound. For example, when training a multi-layer perceptron (one of the most commonly used function approximators) it is common to initialise the network to have random weights. Therefore, during the first few runs of the algorithm the architecture is simply being trained to learn that successive states should have similar nonsense values. Although information is being given to the TDL algorithm, the algorithm is not putting it to any good use at this stage.

Experiments were also made using only the terminal update. In this case the information rate upper bound is $\log_2(3) \approx 1.6$bits, since the only information presented to the algorithm is the outcome of each game (win, lose, or draw).

III. Treasure Hunt Game

Most work on learning to play board games has naturally focused on challenging games that are interesting to play, such as chess, checkers, Othello and go. The disadvantage of these games for our current purpose is that they take a significant amount of CPU time to play, and that the optimal 1-ply value function for each of these games is unknown.

In response to the current need, the author developed the Treasure Hunt Game with the following aims in mind:

- a simple, fast to compute set of rules
- a known optimal policy for each player
- a simple way to encode an optimal policy, using an easily calculated number of bits
- a simply way to vary the size of the game

The Treasure Hunt Game satisfies all these criteria.

A. Rules

Games are always played in sets. A "board" has n squares. Half of them have a value of one to the occupier, the other half have no value. The distribution of values to squares is assigned uniform randomly at the start of each set of games, and remains fixed for that set.

The players have no prior knowledge regarding the value of each square. Each game commences with an empty board. Players take turns to occupy squares with counters of their own colour. Once a square is occupied it stays occupied for the remainder of that game. The game is over when all the squares are occupied. The player who has occupied the most squares of value wins. It's a draw if players have occupied an equal number of valuable squares. Players get no reward signal other than this win/lose/draw signal at the end of the game.

Therefore, players can learn by observing correlations between squares occupied and game outcomes; this is the TDL approach. Alternatively, players can learn by having competing hypotheses about which squares are valuable, and rewarding the hypotheses that tend to lead to game wins,

```
public void inGameUpdate(double[] prev, double[] next) {
    double op = tanh(net.forward(prev));
    double tg = tanh(net.forward(next));
    double delta = alpha * (tg - op) * (1 - op * op);
    net.updateWeights(prev, delta);
}

public void terminalUpdate(double[] prev, double tg) {
    double op = tanh(net.forward(prev));
    double delta = alpha * (tg - op) * (1 - op * op);
    net.updateWeights(prev, delta);
}
```

Fig. 1. The two main methods used to implement temporal difference learning.

by allowing those ones to breed more offspring. This is the evolutionary approach.

A twist to this game is the exploration-exploitation trade-off when playing for a fixed set of games. We ignore this aspect of the game for this paper, and instead focus on the number of games taken to learn the optimal strategy.

B. Optimal Policy

For a game with n squares (i.e. a game of size n) the optimal policy can be represented by a string of n bits where each bit is either zero or one to indicate that a reward is present or not. For learning purposes it is better to allow a measure of confidence or probability that a particular square is valuable, so for all our experiments we use a weighted piece counter as the value function. The agent bases its play on a vector $\mathbf{w}$ of n weights. At each turn in the game, it places a counter of its own colour on the empty square with the highest associated weight. At present this rule is deterministic, but it would be possible to sample from a Boltzmann distribution instead, for example.

A weight vector encodes an optimal policy $\iff$:

$$w_i > w_j \; \forall \; i \in V, j \in \overline{V}$$

where V is the set of all treasure squares, and $\overline{V}$ is the set of all non-treasure squares.

In other words, if all squares of value have higher weights than all squares of no value, then it will play optimally.

These are coded using 8-byte (64 bit) double precision numbers, so technically the number of bits used in each value function is $64n$, but this is highly redundant and the number of bits of information required is simply n. In other words, there are many sets of weights that lead to identical policies.

A multi-valued measure of *true fitness* can also be calculated as follows. This is defined as the number of valuable squares above the median, divided by the number of valuable squares (which is always $n/2$).

IV. Results

In this section the experimental setup is described for each algorithm, then the results are presented and compared. For these experiments the number of games required to learn an optimal weight vector is measured. Recall from the discussion above that any weight vector that assigns higher weights to all valuable squares than to valueless squares encodes an optimal policy. Note however that a TDL player that uses such a weight vector will still play sub-optimally if ϵ, the probability of making a random move, is greater than zero. Hence, for real world learning there would be an exploration-exploitation trade-off problem to be solved. For this paper, we ignore that, and simply measure the number of games taken to learn an optimal weight vector.

In the graphs below all error bars are shown at one standard error from the mean.

A. Evolution

We used two types of evolutionary algorithms. The first was a standard $1 + 9$ evolution strategy (ES), run for 1,000 generations where the fitness was measured by the games won against a random player. Since measuring performance against a random player leads to a noisy fitness function, experiments were conducted to study the effects of varying the number of games played per fitness evaluation. It was found that increasing this figure invariably meant that more games (though fewer fitness evaluations) were needed in order to reach an optimal policy. However, it is important to note that using this sort of algorithm in the presence of noise means that the algorithm does not know when the true optimum has been reached, and it is often reached and then wandered away from. This wandering away from the optimum is possible because a sub-optimal player may still achieve a perfect score against a random player. The algorithm was not penalised for this, and only the first hitting time was measured.

The results for standard evolution are shown in figure 2. The graph only shows board sizes of up to 20. The algorithm failed to reliably learn the optimal policy for larger board sizes (given the population size and number of generations). Various values were tried for m, the number of games per fitness evaluation. Best results (in terms of number of games played to reach the optimum) were obtained with $m = 1$. Note that the algorithm scales very poorly (much worse than linear) with respect to the board size. This is due to the loss of gradient against the fixed random opponent as the evolved player improves.

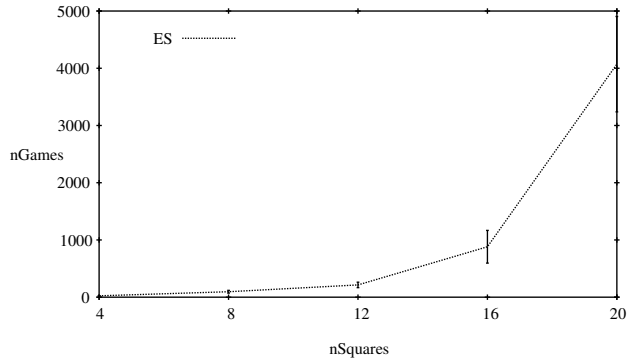

Fig. 2. Number of games against a random player required for standard evolution to learn the optimal policy, plotted against the number of squares on the board.

B. Co-evolution

A similar setup was used for co-evolution, except that the individuals were ranked on their performance in a round-robin league, and the league winner each time was selected as the parent. For the co-evolution experiment, each player played deterministically based on the value of its weight vector, as explained above. Therefore, the co-evolutionary learner had the benefit of noise-free learning (the only noise being in the random mutations made to the weight vectors). Experiments were made with population sizes of 2, 5, and 10 (in each case using a $(1 + (n - 1))$ ES.

The results are shown in Figure 3. The number of games required to reach an optimal weight vector is now approximately linear in the number of games. Furthermore, the upper bounds correctly predicted the rank-order of the algorithms. For smaller population sizes the upper bound is looser.

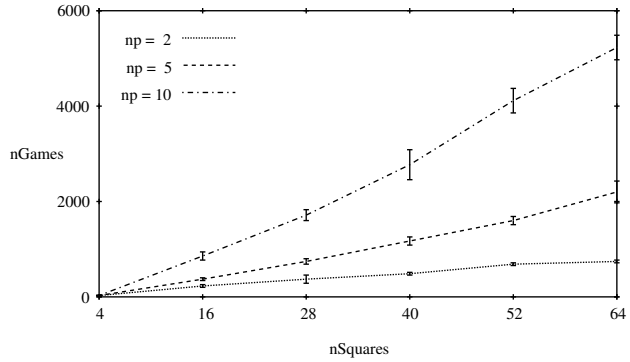

Fig. 3. Number of league games required for co-evolution to learn the optimal policy, plotted against the number of squares on the board.

C. Temporal Difference Learning

TDL was initially used in self-play mode. The setup was taken directly from the work on learning Othello by Lucas and Runarsson [9], as described above in Section II-B, with ϵ set to 0.1, and α set to 0.2. The results are shown in Figure 4. TDL offers much more rapid learning for this game than evolution or co-evolution. To learn optimal play on a board with 64 squares requires only 115 games on average, compared with $1,500$ games for the best performing co-evolution.

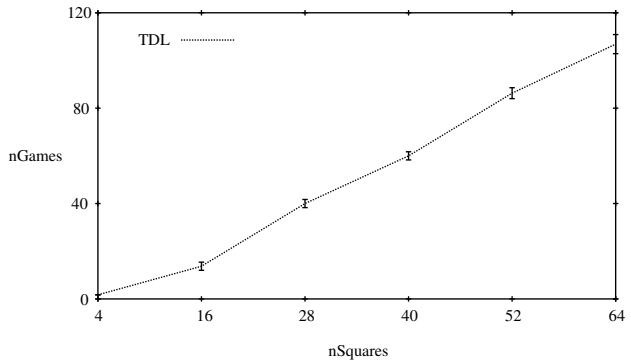

Fig. 4. Number of self-play games required to learn the optimal policy plotted against the number of squares on the board.

Experiments were also made using the TDL player as before, except playing against a random player, rather than using self-play. Interestingly, this greatly diminished the learning ability of the TDL player, unless ϵ was set close to 1.0. This may be because the player could frequently be rewarded for the wrong reasons against a random player, and hence learn a poor policy far removed from the optimum, but still successful enough to defeat a random player most of the time.

One of the simple aspects of the Treasure Hunt Game is that when a counter is placed, it remains unchanged until the end of the game. This suggests that a reinforcement learning algorithm that performed only terminal updates rather than both terminal updates and in-game updates might perform just as well. An experiment was conducted to test this, but it took on average 170 games to learn the optimal policy for a 64 square board. This is significantly more than the 107 required on average when in-game updates were also used.

D. Results Summary

Table II summarises these results, showing information rate upper bounds for each method, average number of games (and standard errors) required to learn optimal strategy for a game size of 64, predicted number of games needed (if the upper bound was reached), and the ratio of actual number of games over predicted number of games.

TDL and TDL' refer to exactly the same algorithm and the empirical results for these use the same data, but the calculations use a different information rate upper bound estimate. The TDL estimate uses in-game and terminal update rates, whereas TDL' is based only on the terminal update.

The first thing to note is that the information rates correctly predict the rank order of the actual games required. Secondly, all estimates apart from the straight TDL estimate are within an order of magnitude of the observed number of games needed, and when it is based on the end-game information

only (TDL') it becomes the closest estimate. Given how over-optimistic the straight TDL estimate is, it maybe that a re-think is required of how the in-game information should be treated.

V. Conclusion

This paper introduced a novel calculation of the upper bounds for information gained during game play for population-based evolutionary learning, and for temporal difference learning.

Two estimates were developed for TDL: one that considers in-game and terminal information, and one that only considers terminal (end-game) information. The one that considers both leads to a terribly over-optimistic estimate on the Treasure Hunt Game. The terminal-estimate is a good predictor of the actual information rate achieved by TDL for this game.

The information rate upper bounds correctly predicted the rank order of observed information acquisition rates, suggesting that this framework provides useful insight into the design of game strategy learning algorithms. In particular, it provides a limit on the total amount of information that can be learned within a given number of games. Even for the simple Treasure Hunt Game a widely used co-evolutionary algorithm took over 700 games to learn 64 bits of information. For more complex games the actual learning rate would most likely be much lower than this.

For both co-evolution and TDL there was a linear relationship between the number of games required to learn the optimal strategy for a given board size of Treasure Hunt Game.

It would be interesting future work to investigate how the rules of the Treasure Hunt Game can be changed to favour either TDL or evolution. This may provide valuable insights into which type of game each class of algorithm is best suited to learn, and how the algorithms might be modified or tuned in order to optimise the information rates. Other important work is to analyse the information rates for more sophisticated evolutionary algorithms such as CoSyNE [10].

References

[1] A. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, pp. 211 – 229, 1959.

[2] D. Michie, "Trial and error," in *In Science Survey, part 2*. Penguin, 1961, pp. 129–145.

[3] G. Tesauro, "Temporal difference learning and TD-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[4] J. Pollack and A. Blair, "Co-evolution in the successful learning of backgammon strategy," *Machine Learning*, vol. 32, pp. 225–240, 1998.

[5] K. Chellapilla and D. Fogel, "Evolving neural networks to play checkers without expert knowledge," *IEEE Transactions on Neural Networks*, vol. 10, no. 6, pp. 1382–1391, 1999.

[6] ——, "Evolving an expert checkers playing program without using human expertise," *IEEE Transactions on Evolutionary Computation*, vol. 5, pp. 422 – 428, 2001.

[7] D. Fogel, *Blondie24: playing at the edge of AI*. Morgan Kaufmann Publishers Inc., 2002.

[8] T. P. Runarsson and S. M. Lucas, "Co-evolution versus self-play temporal difference learning for acquiring position evaluation in small-board go," *IEEE Transactions on Evolutionary Computation*, vol. 9, pp. 628 – 640, 2005.

[9] S. M. Lucas and T. P. Runarsson, "Temporal difference learning versus co-evolution for acquiring othello position evaluation," in *IEEE Symposium on Computational Intelligence and Games*, 2006.

[10] F. Gomez, J. Schmidhuber, and R. Miikkulainen, "Accelerated neural evolution through cooperatively coevolved synapses," *Journal of Machine Learning Research*, vol. 9, pp. 937 – 965, 2008.

[11] C. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, pp. 379 – 423, 623 – 656, 1948.

[12] S. J. Louis and J. McDonnell, "Learning with case injected genetic algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 8, pp. 316 – 328, 2004.

[13] S. J. Louis and C. Miles, "Playing to learn: case-injected genetic algorithms for learning to play computer games," *IEEE Transactions on Evolutionary Computation*, vol. 9, pp. 669 – 681, 2005.

[14] R. Sutton and A. Barto, *Introduction to Reinforcement Learning*. MIT Press, 1998.

Acknowledgements

I thank the anonymous reviewers for their helpful comments on an earlier draft of this paper.

method	bg^{-1}	mean (s.e.)	pred.	ratio
coev (2)	0.500	724 (55)	128	5.7
coev (5)	0.12	2188 (233)	533	4.1
coev (10)	0.037	5223 (276)	1729	3.0
TDL	298	107 (4.5)	0.2	498
TDL'	1.6	107 (4.5)	40	2.7

TABLE II

COMPARISON OF EXPECTED VERSUS ACTUAL NUMBER OF GAMES REQUIRED TO LEARN OPTIMAL POLICY FOR A 64-SQUARE BOARD.