Reinforcement Learning for Games: Failures and Successes

CMA-ES and TDL in comparision

Wolfgang Konen
Cologne University of Applied Sciences,
Faculty for Computer and Engineering Science,
51643 Gummersbach, Germany
wolfgang.konen@fh-koeln.de

Thomas Bartz-Beielstein
Cologne University of Applied Sciences,
Faculty for Computer and Engineering Science,
51643 Gummersbach, Germany
thomas.bartz-beielstein@fh-koeln.de

ABSTRACT

We apply CMA-ES, an evolution strategy with covariance matrix adaptation, and TDL (Temporal Difference Learning) to reinforcement learning tasks. In both cases these algorithms seek to optimize a neural network which provides the policy for playing a simple game (TicTacToe). Our contribution is to study the effect of varying learning conditions on learning speed and quality. Certain initial failures with wrong fitness functions lead to the development of new fitness functions, which allow fast learning. These new fitness functions in combination with CMA-ES reduce the number of required games needed for training to the same order of magnitude as TDL.

The selection of suitable features is also of critical importance for the learning success. It could be shown that using the raw board position as an input feature is not very effective – and it is orders of magnitudes slower than different feature sets which exploit the symmetry of the game. We develop a measure "feature set utility", F_U, which allows to characterize a given feature set in advance. We show that the lower bound provided by F_U is largely in accordance with the results from our repeated experiments for very different learning algorithms, CMA-ES and TDL.

1. INTRODUCTION

Our understanding of learning processes is still limited, especially in complex decision-making situations where the payoff for a particular action occurs only later in time, probably long after the time of the action's execution. The fact that the payoff occurs only after a number of actions leads to the well known *credit assignment problem*: decide, which action should get which credit for a certain payoff. The most advanced methods in machine learning to address this problem are reinforcement learning (RL), e.g., the well-known temporal difference learning (TDL), and evolutionary algorithms, namely evolutionary strategies (ES), and co-evolution.

TDL was applied as early as 1957 by Samuel [12] to the game of checkers and was made more popular through Sutton's work [16, 17] in 1984 and 1988. It became very famous in 1994 with Tesauro's TD-Gammon [19], which learned to play backgammon at expert level.

Another approach to solve RL tasks is neuroevolution where evolutionary algorithms are used to train a neural network. Several algorithms like cellular encoding [1], SANE [11] and NEAT [14] have been proposed which modify the topology and the parameters of the neural networks at the same time. Hansen and Ostermeier proposed in 1996 CMA-ES (Covariance Matrix Adaptation Evolution Strategy) [4] as an alternative neuroevolution approach which optimizes only parameters (weights), not the structure of the net. Igel [5] compares different neuroevolution algorithms on the pole-balancing task and finds CMA-ES to converge faster than the other algorithms.

Board games are a typical source of reinforcement learning problems. Learning game strategy is similar to learning any kind of decision making or control strategy and, since the goal in board games is well defined, they are a good test bed for the development of learning algorithms in general. Although for most board games like chess, checkers or Othello there exist computer programs for expert level play based on carefully designed algorithms, large databases, sheer computing power (or all of this), the task to *learn* such a behaviour in unsupervised form just by letting the computer interact with an environment is only solved for special cases, e.g. Tesauro's TD-Gammon [19]. Currently, a theory on how to design such learning algorithms for *all* kind of games is not known.

Our game scenario in this paper is the game of TicTacToe which has a rather small state space of 5890 states. Why do we use such a simple game and not a more challenging one like chess, Othello or go? – The reasons are similar to those given in [9]: The latter take a significant amount of CPU time to play and the optimal value for most of the states is not known. In contrast, TicTacToe has a small state space, a known optimal policy (which can be easily calculated with the minimax algorithm or value iteration algorithm) and this can be used as a reliable measure of success for a learned agent. If certain settings lead to reliable failure on such a simple game, it is very likely that those settings also do not work for more complex games. In other words, by identifying the terms for failure and success in simple games, we may derive general rules for the design of better learning algorithms on complex games.

The contribution of this paper is to study different approaches to RL. In the scenario described above we search for ingredients which discriminate failure from success in RL. As it is always the case in system theory, the borderline between failure and success is of special interest since the "breakdown" of a system tells the researcher most about such a system. In our case the systems are the reinforcement learning algorithms and we will show some in-

GECCO '09 Montreal, Canada

teresting failures and try to identify in most general terms which elements of the algorithms can turn them into succeses. We demonstrate that providing the right feature set as input to the learner is of utmost importance for all investigated RL algorithms. We derive an empirical formula based on a so-called *feature divergence* to discriminate "good" and "bad" feature sets and show in our experiments that it correctly predicts the rank order of our success measure.

A second goal of this paper is to apply CMA-ES [3, 4] for the first time (at least to our knowledge) to a board game scenario. For game scenarios an optimal fitness function is not easy to define. This is due to the fact that a single game normally covers only a tiny fraction of the game's state space so that success in a single game does not mean that an individual has found the optimal policy. On the other hand, testing an individual by checking its proper reaction on any possible state (i) is prohibitive costly for any normal game, (ii) often the best action for each state is not known and (iii) it can be highly irrelevant if parts of the state space – although legal – are never visited in normal games. We discuss several fitness functions and show that some fitness functions hinder CMA-ES to learn anything at all, at least within the tested number of generations. Others fitness functions allow CMA-ES to learn rather quickly.

Recently, Lucas [9] has given interesting results on comparing the learning rates of TDL and evolutionary algorithms both empirically for a simple game and theoretically with upper bound information rate formulas. We investigate our scenario in a similar direction, i. e. compare between TDL and CMA-ES the computing effort needed for achieving a certain success level. We show results that partly confirm the findings of Lucas and partly extend them since the picture becomes somewhat richer as CMA-ES enters the game.

The rest of this paper is organized as usual: Sec. 2 explains the methods used, namely a short review of TDL and CMA-ES and a comprehensive description of fitness functions, feature sets and success measures used. Sec. 3 applies these method to our board game learning task while Sec. 4 dsicusses these results and draws some conclusions.

2. METHODS

2.1 RL algorithms for TicTacToe

TicTacToe (or Naughts and Crosses) is a simple board game. The board contains 3×3 fields, each player in each move marks (with X or O) a field and the winner is who gets "three in a row" (horizontal, vertical or diagonal). The state space contains 5890 states, among them only 4520 in-game states (the rest are final board positions). This state space is small enough that a standard minimax algorithm can easily perform exhaustive search for each state and find the best move.

A state in strategic games is usually described by the current board position and the player who made the last move (so-called *after state* [18]). An example for TicTacToe is shown in Fig. 1. Following the ideas of Tesauro [19], the RL agent learns the game's *value function* $V(\vec{s}_t)$, which ideally gives for each after state the

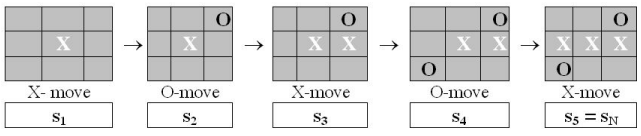

Figure 1: Some after states for the game TicTacToe

probability that player $p = +1$, i.e., "X" will win. Given a certain board position, the strategy for player $p = +1$ is to select the next move which maximizes $V(\vec{s}_{t+1})$, while player $p = -1$ ("O") tries to minimize $V(\vec{s}_{t+1})$. A state can be encoded by collecting row-by-row the board positions into a state vector with $+1$ for each "X", 0 for each unoccupied field and -1 for each "O". If we follow the convention that "X" will always begin, the player who made the last move can be inferred directly from the board position: if the piece count is even it was an "O"-move, otherwise it was a "X"-move. We get for example in Fig. 1 the following state representation for state $\vec{s}_4$, which is a safe win for player "X":

$$\vec{s}_4 = \{\text{00-1}, 011, \text{-100}\}, \qquad V(\vec{s}_4) = 1.000 \tag{1}$$

Not for TicTacToe but usually for most other games the state space is too large to be represented as a table and it is impossible to visit every state sufficiently often during learning. To overcome this problem, a function approximation scheme is used where each state $\vec{s}$ is transformed into a feature state $\vec{g}(\vec{s})$ and the function $f(\vec{w}; \vec{g}(\vec{s}))$ with internal parameter vector $\vec{w}$ (the weight vector) approximates $V(\vec{s})$. We use the same scheme here to make our results more general applicable. Typical approximation functions are

- a linear function $f(\vec{w}; \vec{g}(\vec{s})) = \vec{w} \cdot \vec{g}(\vec{s})$ (or the sigmoid of this linear function)
- a feedforward net with weights $\vec{w}$ and input $\vec{g}(\vec{s})$.

All results in Sec. 3 were obtained with a feedforward net with 15 hidden units and a sigmoidal transfer function.

2.1.1 TDL

The TD algorithm aims at learning the function $f(\vec{w}; \vec{g}(\vec{s}))$. It does so by setting up an (initially inexperienced) RL agent who plays a sequence of games against itself. It learns from the environment which gives a reward $r \in \{0.0, 0.5, 1.0\}$ for { O-win, tie, X-win } at the end of each game. The main ingredient is the *temporal difference* (TD) error signal [18]

$$\delta_t = R(\vec{s}_{t+1}) - V(\vec{s}_t) \tag{2}$$

$$\text{with} \quad R(\vec{s}_{t+1}) = \begin{cases} r(\vec{s}_{t+1}) & \text{if } \vec{s}_{t+1} \text{ is final,} \\ V(\vec{s}_{t+1}) & \text{else} \end{cases}$$

which is used together with the usual gradient descent learning rule to adjust the weights $\vec{w}$. The TD agent is trained in a bootstrapping manner by playing many games against itself. In the initial phase of the training, random moves are added frequently to ensure exploration of the state space. Further details concerning this TD algorithm for games are the same as in Konen and Bartz-Beielstein [8].

2.1.2 CMA-ES

As pointed out by Whitley [20, 21] evolutionary algorithms are especially useful for neural network training when the task at hand is a *reinforcement* learning problem. We use in this work the well-known CMA-ES (Covariance Matrix Adaptation Evolutionary Strategy) based on the work of Hansen and Ostermeier [3, 4] as an evolutionary approach to find optimal weights $\vec{w}$ in our reinforcement learning task. CMA-ES, as Igel [5] points out, (i) allows for direct search in the policy space, (ii) is more robust with respect to the tuning of the algorithm parameters and (iii) has proven to be substantial faster due to small population sizes than other evolutionary algorithms (see [5] for a comparision on the double pole balancing task).

CMA-ES is a rank-based (μ, λ) evolution strategy where the μ best of the λ offspring form the next parent generation. We use in

our experiments the default heuristic to determine the population size from [3]

$$\lambda = 4 + \lfloor 3 \ln(n) \rfloor \quad \text{and} \quad \mu = \lfloor \lambda/4 \rfloor \tag{3}$$

where n is the search space dimension (here: number of weights $\vec{w}$). In CMA-ES each individual represents an n-dimensional real-valued object variable vector. These variables are altered by recombination and mutation. Mutation is realized by adding a normally distributed random vector with zero mean, where the complete covariance matrix is adapted during evolution to improve the search strategy. Recombination is performed globally by computing the center of mass of the μ individuals in the parent generation.

For a more detailed description of CMA-ES the reader is referred to Hansen and Ostermeier [3, 4].

2.1.3 Fitness Functions for CMA-ES

If one applies CMA-ES to a game learning task it is very likely that the first attempts will result in failures. The problem is connected with the proper definition of a fitness function in games. The value function's complete expectation value $\langle V(\vec{s}) \rangle$ (where $\langle \cdot \rangle$ denotes averaging over states $\vec{s}$) would be the ideal fitness in terms of quality. But (i) it is too costly to calculate as fitness, (ii) it is often not known for complex games (or many other decision tasks) and (iii) it is unclear how to weight the different $\vec{s}$ in the expectation value $\langle V(\vec{s}) \rangle$.

Several options can be considered to overcome these problems which lead to different fitness functions, each of them replacing the true expectation value by more or less drastic random sampling:

$F^{(1)}$ Make a fixed choice of of states $\vec{s} \in S$ and benchmark each individual whether it proposes a correct next move (i.e. a move which is as good as the Minimax move) for each element of S. Define the fitness as the percentage of correct decisions. We choose a set S of 48 states which we believe to be relevant, but it is clear that this is only a tiny fraction (about 1%) of the state space. The fitness function is well defined, but does this sample of the state space allow to evolve good players?

$F^{(2)}$ If we do not want to prescribe states we can start with a random population and define the fitness of each individual by the success it has in playing against other individuals from the actual population, either by randomly selecting opponents or by performing a round-robin-league. We choose the first option and furthermore vary each game by starting from one of the 73 possible start positions with 0, 1 or 2 pieces. This fitness seems quite natural for an evolutionary algorithm and it is an option often adopted in the neuroevolution game literature [7], but it has some severe problems which are related to the fact that now the fitness function continuously changes.

$F^{(3)}$ A constant fitness function can be defined as follows: Choose a "good" player as opponent and define fitness as the success rate when playing from one of the randomly selected 73 start positions against this fixed opponent. In the case of TicTacToe we use the Minimax player as the perfect opponent. The randomness in start positions helps to explore the state space so that the evolved player plays strong also against other opponents. Like $F^{(1)}$ the fitness function $F^{(3)}$ is constant during evolution.
Kantschik [6] proposes for the game of chess similarly a constant fitness function which consists however of several human-designed levels with increasingly strong opponents. Moriarty and Miikkulainen [10] use a similar strategy for the game Othello.

$F^{(4)}$ While $F^{(3)}$ turns out to work well in our TicTacToe case, it is not satisfactory because in more complex games a perfect opponent is often not known.[1] Therefore we propose with $F^{(4)}$ a synthesis of $F^{(2)}$ and $F^{(3)}$: Start an evolution (here: CMA-ES) for a first run with a random but fixed opponent, using it in the same way as in $F^{(3)}$. Then replace the opponent with the best individual found so far and restart CMA-ES with the new fitness for the next run. Repeat this procedure until a prescribed number R of runs has been performed. During runs the fitness function remains a constant target, but the mechanism can bootstrap itself to increasingly strong opponents, as we hope, without having such opponents to be prescribed by the user.

2.2 Feature sets

Humans learn complex games by pattern recognition, i.e. by developing specialized feature detectors. Evidently, machine learning algorithms should also make use of features. Some work on game learning [15] incorporates features only indirectly by using only the raw board positions as input and hoping that the neural network in its hidden layer(s) derives the right features. But this might be too difficult in complex games. It is our point that even for simple games, the learning task without features can be orders of magnitudes slower than the one with features, if not being insoluble at all.

Finding valuable features for a learning task is of course not easy. But some aspects are easy to incorporate and can greatly alleviate the learning process. For example the game of TicTacToe has an eightfold symmetry (4 rotations + 4 mirror-rotations). Evidently, it is a waste of computing effort if we present states which are equivalent under this eightfold symmetry as different inputs to the learning algorithm. Features which are invariant under the eightfold symmetry are:

- singlet (one in a row): a line (horizontal, vertical, diagonal) with one piece of either player, the remaining fields being empty;
- doublet (two in a row): the same with two pieces of either player;
- crosspoint: an empty field belonging to at least two singlets of the same player;
- diversity count: the number of different singlet directions for either player;
- corner count: number of corners each player possesses;
- center occupation: -1,0,+1 depending on wheter the center field belongs to `O`, none, `X`.

A crosspoint is of strategic importance: if a player has a crosspoint and can place a piece there, he creates a 'fork', i. e. an opportunity where he can win in two ways. A triplet (three in a row) – although being the goal of the game – is of no importance as feature input since the game is finished as soon as a triplet arises.

Based on these features we form different feature sets as described in Table 1. The features for each set were picked largely at random from the list above. $T1$, $T2$ and $T4$ are invariant under the eightfold symmetry while $T0$ and $T3$ are not. $T3$ is an attempt to enrich the features of $T2$ with the full state information (symmetry lost), while $T4$ is an enrichment which preserves the symmetry.

[1] Likewise, if a "good" opponent were known, it prescribes in an unwanted manner to some extent what the learning algorithm is going to learn.

Table 1: Feature sets for TicTacToe. Each feature vector is an M-dimensional vector: $(t_0, \ldots, t_{M-1})$. $T0$ is not really a feature set but the raw board position given by state $\vec{s}$ as in Eq. (1). It is used as the 'no-feature'-baseline comparision.

Name	Description	dim M
$T0$	$t_{0\ldots9}$: raw board position	9
$T1$	$t_{0,1}$: number of singlets, doublets for O; $t_{2,3}$: number of singlets, doublets for X;	4
$T2$	same as $T1$ plus: $t_{4,5}$: diversity O/X if $p = -1$; 0 else; $t_{6,7}$: diversity O/X if $p = +1$; 0 else; $t_{8,9}$: crosspoint count for O/X;	10
$T3$	$= T2 \cup T0$	19
$T4$	same as $T2$ plus: t_{10} : center occupation; $t_{11,12}$: corner count for O/X;	13

2.2.1 Characterization of Feature Sets: Divergence and Utility

How can we characterize whether a certain feature set is "good" or "bad" for a given task? Usually, the feature space (number of different feature vectors occuring in game play) will be smaller than the state space to make the task somewhat easier for the learning algorithm. But on the other hand the feature space should not oversimplify the situation: It is undesirable that states with different value functions (different final rewards $r = 0, 0.5, 1.0$ assuming perfect play of both sides from that state on) are mapped to the same feature vector. We define a feature vector to be *divergent* if it has this attribute. For a given feature set we define the *divergence measure* Δ as

$$\Delta = 1 - \frac{\sum_{i \in D} N_i}{N} \tag{4}$$

where D is the set of all divergent feature vectors i, N_i is the number of states mapped to the ith feature vector and N is the total number of states. Thus Δ is the probability that a state is mapped to a non-divergent feature vector and $\Delta = 1$ is the best possible value. Of course we have to correct Δ for the probability Δ_0 that a random assignment of states to a feature vector produces a non-divergent feature vector. For example, in the feature set $T0$ each state is mapped to exactly one feature vector (itself), so $\Delta_0 = 1$. If on the other hand, say, 20 states are mapped to one feature vector, then the probability that a random assignment produces a non-divergent feature vector is virtually $\Delta_0 \approx 0$. (This approximation contains the assumption that the states assigned to feature vector i carry each reward with equal probability.) The corrected term $\Delta - \Delta_0$ is one part of the characterization for a "good" feature set.

The second part deals with the number K of different feature vectors occuring in game play. K is measured empirically during game play (game learning) by counting the number of different feature vectors.[2] If we assign an adress to each feature vector i then an upper bound of the information needed to distinguish feature vector i from all others is given by $ld(K)$ (similar to the approach in [9] based on Shannon's information measure [13]). This is an upper bound because for an actual task (like winning the game) some elements of the feature vector may turn out to be irrelevant, thus reducing the effective number K of elements to be distinguished. We propose that the learning task complexity is inversly proportional to that information, leading finally to the following lower bound of *feature set utility*:

$$F_U = 100 \frac{(\Delta - \Delta_0) + a}{ld(K)} \tag{5}$$

where a is an empirical factor accounting for the fact that even for $\Delta - \Delta_0 = 0$ (e. g. the cases $T0$ and $T3$) a certain feature set may well contain learnable information. Here we take $a = 0.3$, but the results are not very sensitve on a.

[2]In this way we count only the feature states actually occuring in game play, not the theoretically possible feature states.

Table 2: Feature set utility and divergence measure: For the feature sets of Table 1 with dimension M and number K of different feature vectors we show their divergence measures Δ and their feature set utility F_U from Eq. (5) for $a = 0.3$.

Name	M	K	Δ	$\Delta - \Delta_0$	F_U
$T0$	9	4501	100%	0%	2,5
$T1$	4	111	33%	33%	9,2
$T2$	10	121	74%	74%	14,4
$T3$	19	4501	100%	0%	2,5
$T4$	13	486	95%	95%	13,2

Our hypothesis is that feature sets with large utilities F_U (shown in Table 2) should make the learning task considerably easier. $T2$ and $T4$ have a large F_U since they provide a large $\Delta - \Delta_0$ *and* at the same time a feature space much smaller than the original state space. Note that F_U is a lower bound, as can be seen from feature set $T3$: $T3$ contains $T2$ as a subset, so if a learning algorithm learns to ignore the extra $T0$-inputs, $T3$'s utility can be eventually as large as $T2$'s utility. However, the learning algorithm has first to learn to ignore the $T0$-inputs, so we expect $T3$'s utility to be somewhere between 2,5 and 14,4.

2.3 Success Measures

For board games it is a bit difficult to define when a player is really successful or optimal due to the following reasons: (i) the optimal policy for each possible state is often not known or too costly to calculate and (ii) the real performance in a league largely depends on the environment, i. e. the strength of the opponents. Repeated games only against the optimal player (Minimax in the case of TicTacToe) might explore only a tiny fraction of the state space. We therefore define a success measure based on three elements

- $S^{(rand)}$: success rate when playing 100 game doubles against RandomPlayer, which draws its moves completely at random. A game double consists of two games where each player assumes both roles, X and O. The success is measured as -1, 0, 1 for win, tie, loss. It turns out, that even the optimal player Minimax cannot achieve $S^{(rand)} > 0.9$ on average, since the random player will get a tie in 20% of the games when playing X.
- $S^{(mini)}$: success rate when playing a game double against Minimax, the perfect player. Since no one can win against Minimax, the best possible score is $S^{(mini)} = 0$, the worst is -1.
- $S^{(val)}$: success rate when playing 100 game doubles against another opponent (based on value iteration), which plays as strong as Minimax but uses different moves and selects randomly among them when there are moves of equal strength. Again the best possible score is 0, the worst is -1.

Since some learning algorithms might be biased towards a single success measure as they might use it directly or indirectly during

Table 3: Success or failure of different algorithms on the TicTacToe learning task. All experiments used feature set $T2$ and neuroevolution of a feedforward net having one hidden layer with 15 neurons. In columns 2 and 3 we show the number G of games needed on average to reach the condition of row 2 for the success measure p_S. Column 4 shows the average value of p_S after 10^5 games. A dash in the columns for G means that the condition was not met within 3×10^5 games.

Quantity	G	G	p_S
Condition	$p_S \geq 60\%$	$p_S \geq 80\%$	$G = 10^5$
Algorithm			
CMA-ES with $F^{(1)}$	-	-	50%
CMA-ES with $F^{(2)}$	30000	-	55%
CMA-ES with $F^{(3)}$	3000	3000	100%
CMA-ES with $F^{(4)}$	31200	120000	70%
CMA-ES with $F^{(4)}$, $n^{(Pool)} = 3$	7200	30000	96%
RWG	-	-	0%
TDL	5000	10000	100%

learning, we define an overall success of a learned player as the combination

$$S = (S^{(mini)} = 0) \wedge (S^{(rand)} > 0.8) \wedge (S^{(val)} > -0.1) \quad (6)$$

From our experiments we know that a player fulfilling all three criteria will play perfect in nearly all constellations. In our experiments described below we often compute Z different realizations of the same learning task to get statistically sound results. As a measure which is not disturbed by single outliers we propose

$$p_S = R_S/Z, \quad (7)$$

the percentage of realizations which are successes according to Eq. (6), where R_S is the number of realizations with S fulfilled. This success measure characterizes the certainty with which a training run will produce a strong player.

A second combined success measure

$$S_C = 1 + \frac{(S^{(rand)} - 0.9) + S^{(mini)} + S^{(val)}}{3} \quad (8)$$

delivers a real number between 1 (perfect) and 0 (bad). It allows to calculate standard deviations from repeated experiments.

3. RESULTS

3.1 Failures due to Fitness Functions

We applied CMA-ES in its standard settings to the TicTacToe learning task. The fitness functions $F^{(2)}$, $F^{(3)}$, $F^{(4)}$ were the success rate in a tournament of $n_G = 30$ game doubles against a (fixed or randomly selected) opponent where in each game double one of 73 starting positions (see Sec. 2.1.3) was randomly selected. Each experiment was repeated $Z = 25$ times with new random weights to measure the statistical fluctuations.

It is very convenient that CMA-ES is a (or can be used as a) parameter-free optimization algorithm. However, the first two fitness functions $F^{(1)}$ and $F^{(2)}$ and in part also the first realization of $F^{(4)}$ were complete failures, see Table 3, at least within the number of games tested. (For $F^{(1)}$ there are strictly speaking no games to count, but we translate the number of necessary move evaluations into games using the average number of 7 moves/game.)

For $F^{(1)}$ this is not a big surprise since a tiny fraction of the state space is not likely to produce good game players. In our experiments we saw that the population reaches very quickly a plateau where all individuals have a perfect fitness $F^{(1)}$ so there is nothing to learn for them further, although they have not yet learned a good strategy.

It is somewhat more surprising that $F^{(2)}$ turns out to be a failure since one could expect that playing against other individuals should eventually strengthen the population. But this strategy turns out to work not for CMA-ES and it is also likely that it is the reason for other partial failures reported in the literature (e. g. [7]): A closer inspection reveals that the always-changing nature of the fitness function $F^{(2)}$ is the problem: In the beginning virtually all individuals are weak. If one individual is slightly better than the others, it will score a high fitness because it can beat all others. This is not likely to happen again later in the training, so the fitness function becomes poorer although the players are stronger. Nevertheless, the best-ever solution returned is one from the initial phase resulting in a mediocre player. We find these hypothesis verified in our CMA-ES runs since with fitness $F^{(2)}$ the evolution process reports always the best-ever solution to occur somewhere in the first 20 generations.

There are two options to overcome this weak fitness $F^{(2)}$: to make a constant fitness $F^{(3)}$ with an opponent delivered from outside the learning algorithm (Minimax in the case of $F^{(3)}$) or to work with evolution restarts and to keep between restarts the opponent fixed (coming from the popultion set as described with fitness $F^{(4)}$ in Sec. 2.1.3).

We find $F^{(3)}$ to work very well as can be seen from Table 2: it reaches within 10 generations or 3000 games a 80%-level for p_S which $F^{(1)}$ and $F^{(2)}$ never achieve and it quickly boosts this up to 100% if we increase the number of generations. Is the task perhaps with this fitness and this feature set so easy that any learning algorithm can learn it? – We tested this hypothesis by running the same experiment having only CMA-ES replaced by *Random Weight Guessing* (RWG): Each weight is drawn randomly from the uniform distribution $[-1, 1]$. As Table 2 shows, RWG does not produce a single good player at all, so the task is not trivial.

$F^{(4)}$ on the other hand is in its first attempt only partially satisfactory (see Table 2): It reaches the ($p_S = 80\%$)-level only after 120000 games (200 generations) and even 300000 games will not bring p_S above the 90%-level. Results were obtained with $R = 4$ restarts (we tested also $R = 8$ and $R = 12$ but found the results to be rather similar).

Again, this failure is instructive because a closer inspection of the fitness curves for a single training shows: a player which finally failed on the p_S-criterion was often successful before prior restarts. That is, a success level once reached was lost when changing the opponent. We attribute this to the fact that the learned CMA-ES mean and CMA-ES distribution were no longer useful when switching to a different opponent. We therefore modified the fitness function to become less discontinuous between restarts: Instead of a single opponent we used a pool of $n^{(Pool)} = 3$ opponents where on a restart only *one* of those opponents was replaced in a round-robin fashion. Algorithm 1 gives the detailed pseudocode for this procedure. The result with $n^{(Pool)} = 3$ in Table 2 shows that this brings a spectacular improvement compared to the baseline $F^{(4)}$ (with $n^{(Pool)} = 1$): the number of games needed to reach a medium p_S-level drops by a factor of *four* and the final p_S-level is only 4% below the optimum.

Finally we compare results from CMA-ES runs to those from TDL runs, the other well-known learning method for reinforcement tasks. The TDL algorithm and its parameter settings are the same

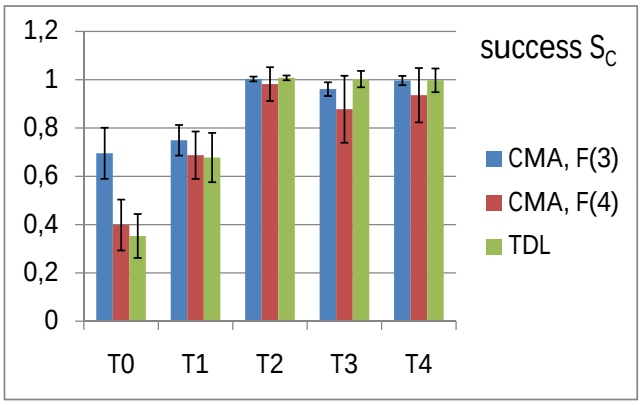

Figure 2: Success S_C acc. to Eq. (8) for the different algorithms. Each bar represents the average of 25 realizations based on $G = 60000$ games of training using the feature set as indicated on the horizontal axis. Error bars indicate one standard deviation

as described in [8]. As Fig. 2 shows, all algorithms are comparable on feature set $T2$ while there is a slight advantage of CMA$[F^{(3)}]$ and TDL over CMA$[F^{(4)}]$ on feature sets $T3$ and $T4$. We see from Table 3 that TDL is both in quality and in time (number G of games needed) comparable to the two best CMA-ES results $F^{(3)}$ and $F^{(4)}(n^{(Pool)} = 3)$.

It is interesting to compare this with Lucas' findings [9] who compared TDL and co-evolution on another simple game and found that TDL had learning rates by a factor of 7 to 50 faster than co-evolution. We find here that CMA-ES brings the learning rate of an evolutionary algorithm into the same region as TDL, but only if we use well-designed fitness functions. Thus CMA-ES has, due to its directed mutations, the potential to increase the information rate in the sense as discussed by Lucas [9].

3.2 Failures due to Wrong Features

How is the performance of different feature sets on the same learning task? Are the conclusions drawn with respect to features similar when different learning algorithm are used? We show in Fig. 3 and Fig. 4 the performance of different feature sets using CMA-ES (with the two best fitness functions found in Sec. 3.1) and using TDL.

Most notably, the raw board position input (feature set $T0$) and the feature set $T1$ which contains only 4 features, are complete failures for every learning algorithm. Stenmark [15] obtained some good results with the raw board position as input to a similar feed-forward net, but only after *one million* of games. In our experiments with up to 300000 games we could not see any learning progress.

In contrast to that, feature sets $T2$ and $T4$ allow successful CMA-learning ($p_S > 90\%$) as early as after 6000 and 12000 games with fitness function $F^{(3)}$. This is an increase in learning rate by a factor of 160 and 80, resp., compared to [15]. TDL performs similarly to CMA$[F^{(3)}]$ with the feature sets $T2$ and $T4$ crossing the ($p_S > 90\%$)-level after 9000 and 17000 games (Fig. 4).

It is interesting to see that all three diagrams in Fig. 3 and Fig. 4 show a strict ranking order

$$p_S(T2) \geq p_S(T4) \geq p_S(T3) \geq p_S(T1) \geq p_S(T0)$$

for all measurements (with the sole exception of the $G = 100000$ experiment for TDL where $T4$ and $T3$ are interchanged). This is

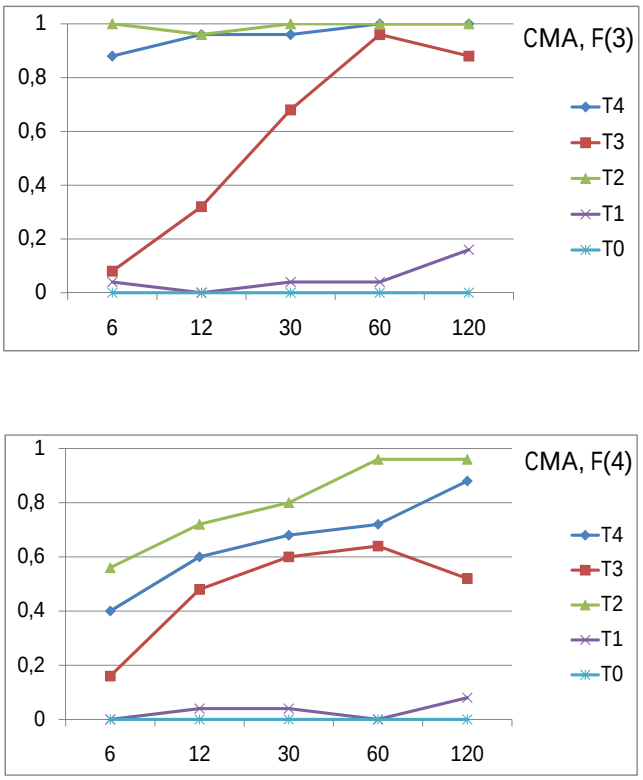

Figure 3: Comparision of different feature sets: Shown is the success rate p_S (percentage of successful players out of 25 realizations, acc. to Eq. (7)) as a function of the number of games G divided by 1000. All results were obtained with CMA-ES used for neuroevolution of a feedforward net having one hidden layer with 15 neurons. The upper diagram uses fitness function $F^{(3)}$ and the lower uses fitness $F^{(4)}$ with $R = 4$ and $n^{(Pool)} = 3$.

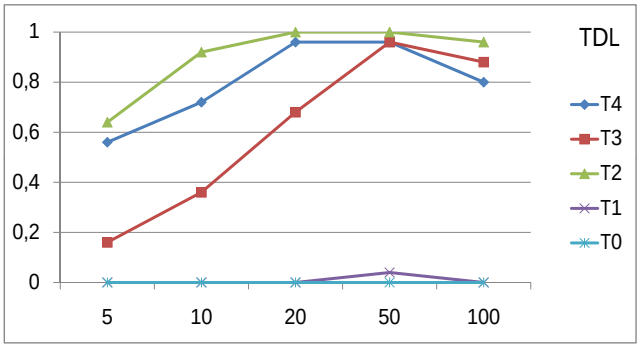

Figure 4: Comparision of different feature sets: Same as Fig. 3 but with TDL instead of CMA-ES.

in good agreement with the ranking order of the feature set utility F_U which reads according to Table 2

$$F_U(T2) \geq F_U(T4) \geq F_U(T1) \geq F_U(T3) \geq F_U(T0)$$

We have only a one-place disagreement between $T3$ and $T1$. But this does not come unexpected, as already explained in Sec. 2.2.1, since F_U is only a lower bound and the true utility for $T3$ can be somewhere between $F_U(T0) = 2.5$ and $F_U(T2) = 14.2$. $T3$ reaches the upper bound if the learning algorithm learns to ignore the $T0$-part of the $T3$-inputs. This view is supported by our experiments: The diagrams for TDL and for CMA$[F^{(3)}]$ show that $T3$ takes considerably longer to achieve good performance (it has to learn to ignore the 'distractive' $T0$-part) but eventually reaches the same performance as $T2$ at 50000 and 60000 games, resp.

In the case of CMA$[F^{(4)}]$ the 'distractive' $T0$-part in $T3$ seems to be more confusing for the learning algorithm, since it does not reach the same performance as $T2$. Another point which is not yet fully understood is the fact, that all learning algorithm show for $T3$ a decline in performance as we go beyond G=60000 games.

It may be somewhat surprising that $T4$, although being 'richer' than $T2$ and obeying in the same way the eightfold symmetry, is always inferior to $T2$. But the point is, as also correctly predicted by our feature set utility F_U, that $T4$ has a higher dimension and much higher feature vector count K which counteracts the beneficial effect of a lower divergence (higher $\Delta - \Delta_0$).

In summary we have demonstrated the great importance of features for our application. We were able to develop with the feature set utility F_U a qualitative measure based on information-theoretic and empirical ingredients which is able to correctly predict the ranking order of the different feature sets' performances. This ranking order holds for very different RL learning algorithms TDL and CMA-ES.

4. CONCLUSION

CMA-ES has been applied for the first time, at least to our knowledge, to board game learning based on RL. Initially some instructive failures were observed which shed some light on the importance of the right fitness function. Some lessons learned:

- Do not rely on only a subset of board positions as fitness function ($F^{(1)}$). CMA-ES will produce agents which fulfill the fitness function perfectly, but they are usually "blind" for other board positions.
- Do not allow the fitness function to be dependent on the population ($F^{(2)}$). If fitness is defined as success against others from population, then initial results against a weak population will very likely mask later improvements and hinder CMA-ES to learn.

But if we use CMA-ES with a good fitness function – either a constant fitness like $F^{(3)}$ or a bootstrap version $F^{(4)}[n_{Pool} = 3]$ with not too abrupt changes between restarts – then CMA-ES learning turns out to be successful, even comparable to TDL in terms of learning speed in the case of $F^{(3)}$. Since Lucas [9] reports other evolutionary learning schemes like co-evolution to be much slower than TDL, we conclude that CMA-ES has the potential to learn considerably more from each game in comparision to co-evolution.

Not surprisingly, feature selections turns out to be another critical factor for distinguishing failures from successes: raw inputs or too simple feature sets lead to consistent failures in a variety of learning algorithms, while other feature sets ($T2$, $T4$ and $T3$ in this order) lead to more or less fast learning. There is not yet a comprehensive theory on how to find the best feature set, but some lessons can be learned from our experiments:

- Find features which obey the symmetries of the game, this will lead to better generalization.
- Find feature sets which combine two antagonistic elements as good as possible: (i) a low count K of different feature vectors during game play and (ii) a low divergence (high $\Delta - \Delta_0$) within the states mapped to the same feature vector.

A general recipe for finding feature sets which balance both elements is not yet known. But we have introduced with the feature set utility F_U of Eq. (5) a measure which allows to give at least a lower bound on the usefulness of a certain feature set. F_U has shown to be in good accordance with the results from our board game scenario.

Further directions of research remain to be done: Can the feature set utility F_U prove to be useful also for other, more realistic games like Othello or Connect-4? Can a general recipe be developed on how to find or 'evolve' good feature sets? Is the bootstrapping CMA-ES approach also successful for larger games? We hope to adress some of these questions in the near future, perhaps again being led by some instructive failures.

5. ACKNOWLEDGMENTS

This work was supported in part by COSA, a research center grant from the Cologne University of Applied Sciences. The implementation of CMA-ES was directly taken from the freely available source code by Nikolaus Hansen [2] in Java which is a highly recommendable and well documented software. Thanks!

6. REFERENCES

[1] F. Gruau, D. Whitley, and L. Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In J. R. K. et al., editor, *Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, CA, USA*, pages 81–89, 1996.

[2] N. Hansen. CMA evolution strategy source code. `http://www.lri.fr/~hansen/cmaes_inmatlab.html`, 2008.

[3] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9:159–195, 2001.

[4] N. Hansen, A. Ostermeier, and A. Gawelczyk. Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. In *Proc. IEEE International Conference on Evolutionary Computation*, pages 312–317. Morgan Kaufmann, 1996.

[5] C. Igel. Neuroevolution for reinforcement learning using evolution strategies. In R. S. et al., editor, *Proc. Congress on Evolutionary Computation (CEC 2003)*, pages 2588–2595, 2003.

[6] W. Kantschik. *Genetische Programmierung und Schach*. PhD thesis, Universität Dortmund, Germany, 2006.

[7] K.-J. Kim and S.-B. Cho. Systematically incorporating domain-specific knowledge into evolutionary speciated checkers players. *IEEE Transactions on evolutionary computation*, 6:615–627, 2005.

[8] W. Konen and T. Bartz-Beielstein. Reinforcement learning: Insights from interesting failures in parameter selection. In G. Rudolph, editor, *10th International Conference on Parallel Problem Solving From Nature (PPSN2008)*, pages 478–487. Springer, Berlin, 2008.

[9] S. Lucas. Investigating learning rates for evolution and temporal difference learning. In *Proc. IEEE Symposium on Computational Intelligence and Games CIG2008*, Perth, Australia, December 2008. IEEE Press.

[10] D. E. Moriarty and R. Miikkulainen. Discovering complex othello strategies through evolutionary neural networks. *Connection Science*, 7:195–209, 1995.

Algorithm 1 CMA-ES with opponent pool and restarts as a learning algorithm for strategic games. A CMAPlayer is a trainable player for the game in question, its value function approximator being trained by CMA-ES. The algorithm reduces for $R = 1$ to the usual CMA-ES.

Input: number g of CMA-generations, number R of runs, size $n^{(Pool)}$ of the opponent pool and fitness F which is the average success (games won) against the opponent pool.

Initialize CMAPlayer, initialize its opponent pool `O[]` with $n^{(Pool)}$ CMAPlayers having random weights. Set `s=0`.
for $(i = 0; i < R; i++)$ **do** ▷ The sequence of generations is divided into R runs.
    Set `maxGen` $= \lfloor (i+1) * g/R \rfloor$
    Evolve CMAPlayer with CMA-ES using fitness F, until `maxGen` generations are reached or until another stop condition is met.
    Set B = best-ever individual in this run and μ = mean of actual CMA-population.
    if $i < R - 1$ **then**
        Replace one of the opponents, `O[s]`, from the pool with the best individual B.
        Recalculate $F(B)$ and $F(\mu)$ with the *altered* pool. The better of both becomes the new best-ever solution.
        `s = (s+1) %` $n^{(Pool)}$ ▷ Select the next-in-turn opponent.
    end if
end for
Return the best-ever solution B (from the last run $i = R - 1$)

[11] D. E. Moriarty and R. Miikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22:11–32, 1996.

[12] A. Samuel. Some studies in machine learning using the game of checkers. *IBM journal of Research and Development*, 3(3):210–229, 1959.

[13] C. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 1948.

[14] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[15] M. Stenmark. Synthesizing board evaluation functions for Connect4 using machine learning techniques. Master's thesis, Østfold University College, Norway, 2005.

[16] R. S. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst, MA, 1984.

[17] R. S. Sutton. Learning to predict by the methods of temporal differences. In *Machine Learning*, pages 9–44, 1988.

[18] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[19] G. Tesauro. TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.

[20] D. Whitley. Genetic algorithms and neural networks. In *Genetic algorithms in Engineering and Computer Science*, chapter 11. Wiley, 1995.

[21] D. Whitley, S. Dominic, R. Das, and C. Anderson. Genetic reinforcement learning for neurocontrol problems. *Machine Learning*, 13:259–284, 1993.