Reinforcement Learning with N-tuples on the Game Connect-4

Markus Thill, Patrick Koch and Wolfgang Konen*

Department of Computer Science, Cologne University of Applied Sciences,
51643 Gummersbach, Germany,
{patrick.koch, wolfgang.konen}@fh-koeln.de

Abstract. Learning complex game functions is still a difficult task. We apply temporal difference learning (TDL), a well-known variant of the reinforcement learning approach, in combination with n-tuple networks to the game Connect-4. Our agent is trained just by self-play. It is able, for the first time, to consistently beat the optimal-playing Minimax agent (in game situations where a win is possible). The n-tuple network induces a mighty feature space: It is not necessary to design certain features, but the agent learns to select the right ones. We believe that the n-tuple network is an important ingredient for the overall success and identify several aspects that are relevant for achieving high-quality results. The architecture is sufficiently general to be applied to similar reinforcement learning tasks as well.

Keywords: Machine learning, reinforcement learning, TDL, self-play, n-tuple systems, feature generation, board games

1 Introduction

1.1 Learning

Our understanding of learning processes is still limited, especially in complex decision-making situations, where the payoff for a particular action occurs only later, probably a long time after the action is executed. The fact that the payoff occurs only after a number of subsequent 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 evolution strategies (ES) and co-evolution.

TDL was applied as early as 1957 by Samuel [9] to checkers and gained more popularity through Sutton's work in 1984 and 1988 [13, 14]. It became

* This work has been partially supported by the Bundesministerium für Bildung und Forschung (BMBF) under the grant SOMA ("Ingenieurnachwuchs" 2009).

very famous in 1994 with Tesauro's TD-Gammon [15], which learned to play backgammon at expert level.

Learning to play board games has a long tradition in AI. This is due to most board games having simple rules, nevertheless, they encode surprisingly complex decision-making situations including the above mentioned credit assignment problem. The search for an optimal playing agent constitutes an interesting branch of optimization problems: In normal optimization problems, an objective function is known, i. e. a function to assess the quality of a solution. Contrariwise, no such objective function exists for many board games, or it is computationally too expensive to calculate. For example, most board positions in chess are too complex to be analyzed in full depth. Moreover, the strength of an agent is the quality of its move in *all* relevant board positions. Instead of this inaccessible objective function, we often use certain *interactions* as primary driver of the search, which serve as a surrogate for the objective function [7]. On a simple level, interaction can take place between a solution and itself: A well-known example is self-play, a technique used for game strategy learning, where an agent plays many games against itself. In population-based methods like co-evolutionary algorithms, interaction involves two or more different solutions from one or from several populations.

In particular, we are interested in such learning problems, where a 'true' objective function is not accessible for learning. Many problems in practice are like this: We need to 'play well' in a certain environment, but the objective function is either not known or it may not be invoked often enough for all the learning trials needed. Given the right learning architecture, how much can we learn from self-play? Nature itself provides us with incredibly convincing examples, for instance, children, who learn in complex situations from only few interactions with the environment. It is an open question, how much internal self-play contributes to such learning successes. Our goal is to mimic at least some of these awesome learning successes observed in nature with machine learning architectures.

1.2 Related Work

In our previous work concerned with the learning of game functions, we compared CMA-ES and TDL [6] and investigated the role of features and feature generation on learning and self-organization [5].

In this paper we consider the game Connect-4 as a specific example, which is solved on the AI-level (see Sec. 2.1). Only rather few attempts to *learn* Connect-4 (whether by self-play or by learning from teachers) are found in the literature: Schneider et al. [10] tried to learn Connect-4 with a neural network, using an archive of saved games as teaching information. Sommerlund [11] applied TDL to Connect-4 but obtained rather discouraging results. Stenmark [12] compared TDL against a knowledge-based approach from Automatic Programming and found TDL to be slightly better. Curran et al. [3] used a cultural learning approach for evolving populations of neural networks in self-play to play Connect-4. All the above works gave no clear answer on the playing strength of the agents,

since they did not compare their agents with a perfect-playing Minimax agent. Some preliminary work in our institution on learning Connect-4 with neural nets or TDL also did not lead to convincing success. Only small subsets of Connect-4 could be learned to some extent.

Then Lucas showed with the n-tuple-approach [8] that the game of Othello, having a somewhat greater complexity than Connect-4, could be learned by TDL within a few thousand training games. The n-tuple-approach is basically a clever approach to introduce a rich variety of features into board games. Krawiec et al. [7] applied the n-tuple-approach in (Co-) Evolutionary TDL and outperformed TDL in the Othello League. This stirred new interest in our Connect-4 project and gave rise to the following research questions:

Q1 Is the n-tuple approach also applicable to Connect-4, i. e. can we, for the first time, get good results from self-play using n-tuples?

Q2 Can we do so with relatively *few training games* when using enough n-tuples?

Q3 Which ingredients are crucial for the success of learning?

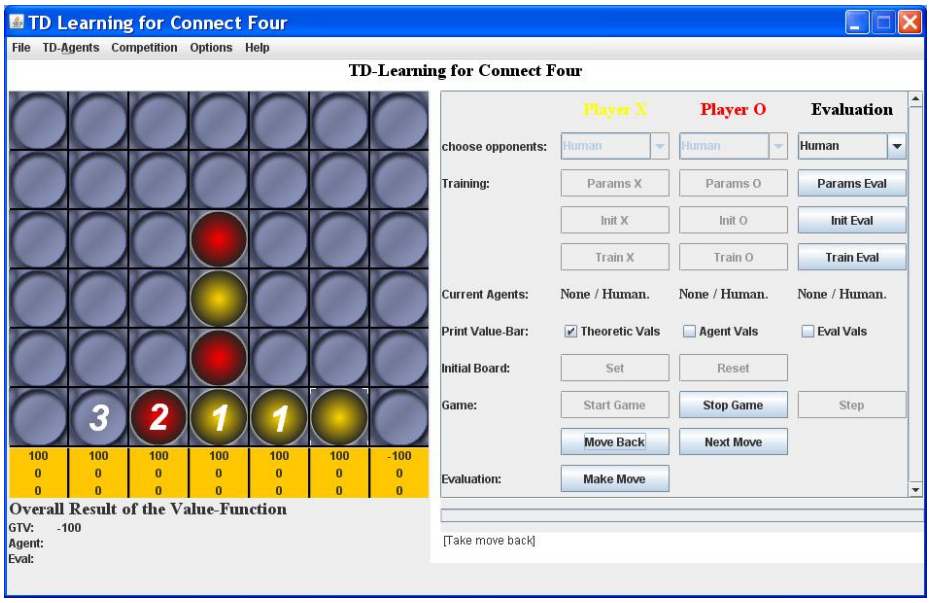


Fig. 1. Connect-4 board with an example 4-tuple '3-2-1-1' (see Sec. 2.2)

2 Methods

2.1 Connect-4

The game of Connect-4 is a two-player game played on a board with 7 vertical slots containing 6 positions each (Fig. 1). Player Yellow (1st) and player Red (2nd) place one piece per turn in one of the available slots and each piece falls down under the force of gravity into the lowest free position of the slot. Each

player attempts to create horizontal, vertical or diagonal piece-lines of length four. Fig. 1 shows an example position where Yellow would win if Red does not block this by placing a red piece into the right slot.

Connect-4 has a medium state space complexity of $4.5 \cdot 10^{12}$ board positions [4]. It can be solved by a combination of game tree search and clever heuristics with a few days of computing time. Connect-4 was solved in 1988 independently by Allen and by Allis [1]: Yellow (the 1st player) wins, if she places her piece in the middle slot.

We developed a Minimax agent combined with a pre-calculated 8-ply or 12-ply-opening database [16] and it finds the perfect next move (or moves, if several moves are equally well) for each board position within fractions of a second. This agent will be used in our experiments only as referee or as evaluating agent. It is by no means used for any training purpose.

2.2 N-tuples and LUTs

N-tuples in General N-tuple systems have first been introduced in 1959 by Bledsoe and Browning [2] for character recognition. Their main advantages include conceptual simplicity and capability of realizing non-linear mappings to spaces of higher dimensionality [7]. Recently, Lucas proposed employing the n-tuple architecture also for game-playing purposes [8]. The n-tuple approach works in a way similar to the kernel trick used in support vector machines (SVM): The low dimensional board is projected into a high dimensional sample space by the n-tuple indexing process [8].

N-tuples in Connect-4 Each n-tuple T_i is a sequence $[a_{i0}, \ldots, a_{in-1}]$ of n different board locations where each a_{ij} codes a specific cell of the board. For example, the four white digits in Fig. 1 mark the board locations of a 4-tuple. Each location possesses one of P possible states $z[a_{ij}] \in \{0, \ldots, P-1\}$ (the value of the digits in our example). An n-tuple of length n thus has P^n possible states $k \in \{0, \ldots, P^n - 1\}$. The number represented by the state of T_i can be used as an index into an associated look-up table LUT_i, which contains parameters $w_{i,t}[k]$ equivalent to weights in standard neural networks. For a given board position z, the output of the n-tuple network can be calculated as:

$$f(\boldsymbol{w}_t, z_t) = \sum_{i=0}^{m} w_{i,t}\,[k] \quad \text{with} \quad k = \sum_{j=0}^{n-1} z_t[a_{ij}] P^j. \tag{1}$$

Here, $z_t[a_{ij}]$ is the state of board location a_{ij} at time t. Likewise, $w_{i,t}[k]$ is the weight for state k of n-tuple T_i, $i = 1, \ldots, m$. It is also a function of time t since it will be modified by the TD learning rule, see Sec. 2.3. The vector $\boldsymbol{w}_t$ combines all weights from all LUTs at time t.

Position Encoding We consider two alternative approaches for Connect-4:

- **P=3:** Each board location has one of the 3 z-values: 0=empty, 1=Yellow, 2=Red.
- **P=4:** Each board location has one of the 4 z-values: 0=empty and not reachable, 1=Yellow, 2=Red, 3=empty and reachable.

By *reachable* we mean an empty cell that can be occupied in the next move. The reason behind the (P=4)-encoding is that it makes a difference whether e. g. three yellow pieces in a row have a reachable empty cell adjacent to them (a direct threat for Red) or a non-reachable cell (only indirect threat).

Fig. 1 shows an example board position with a 4-tuple in the numbered cells. The state of the 4-tuple is $k = 3 \cdot 4^0 + 2 \cdot 4^1 + 1 \cdot 4^2 + 1 \cdot 4^3 = 91$.

Symmetry Board games often have several symmetries in the board position. In case of Connect-4, there is only one symmetry, the mirror reflection of the board along the middle column. If k denotes an n-tuple state for a certain board position, we denote with $M(k)$ its state in the mirror-reflected board. As proposed by Lucas [8] we can augment the n-tuple network output of Eq. (1) to

$$f(\boldsymbol{w}_t, z_t) = \sum_{i=0}^{m} (w_{i,t}[k] + w_{i,t}[M(k)]). \tag{2}$$

N-tuple Creation There are different methods how n-tuples can be created: The purely random sampling of board cells is possible but not very sensible for game feature search. It is more advisable to select adjacent locations because the goal of the game is to place 4 adjecent pieces. Therefore, we propose a random walk very similar to the snake method introduced by Lucas [8]: To create an n-tuple of length K, we start at a random cell, which is the first cell of the n-tuple. Afterwards, we visit one of its 8 neighbors and add it to the n-tuple (if not already in there). We continue to one of its neighbors, and so on, until we have K cells in the n-tuple. The only difference to Lucas' snake method is our n-tuple all having a fixed length of K, while snakes can have lengths $2, \ldots, K$.

2.3 TDL

The goal of the agent is to predict the ideal *value function*, which usually is 1.0 if the board position is a win for Yellow, and -1.0 if it is a win for Red. The TD algorithm aims at *learning* the value function. It does so by setting up an (initially inexperienced) agent, who plays a sequence of games against itself. It learns from the environment, which gives a reward $r \in \{-1.0, 0.0, 1.0\}$ for { Yellow-win, Draw, Red-win } at the end of each game . The main ingredient is the *temporal difference* (TD) error signal [14]

$$\delta_t = V(\boldsymbol{w}_t, z_{t+1}) - V(\boldsymbol{w}_t, z_t). \tag{3}$$

Here, $V(\boldsymbol{w}_t, z_t) = \sigma\left(f(\boldsymbol{w}_t, z_t)\right)$ is the agent's current approximation of the value function on the basis of Eq. (2) and a nonlinear sigmoid function σ (we choose $\sigma = \tanh$). If the state observed at time t + 1 is terminal, the exact value r of the game is used in Eq. (3) instead of the prediction $V(\boldsymbol{w}_t, z_{t+1})$. The weights are trained with the usual δ-rule

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \delta_t \boldsymbol{\nabla}_w V(\boldsymbol{w}_t, z_t), \tag{4}$$

which aims at making the preceding prediction match the current prediction more closely. More details on TDL in games can be found in our previous work [5].

Table 1. Computation time, number of weights and (for a specific n-tuple set) non-zero weights. The count of non-zero weights varies a bit with different n-tuple sets, but always has the same order of magnitude.

position encoding	time (10× 10^7 games)	weights	non-zero weights
P=3	6h 45min	918.540	304.444
P=4	7h 50min	9.175.040	646.693

2.4 Agent Evaluation

A fair agent evaluation for board games is not trivial. There is no closed-form objective function for 'agent playing strength' since the evaluation of all board positions is infeasible and it is not clear, which relevance has to be assigned to each position. The most common approach is to assess an agent's strength by observing its *interactions* with other agents, either in a tournament or against a referee agent. We choose the Minimax agent to be the ultimate referee. Note that all the approaches to Connect-4 found in the literature (cf. Sec. 1) fail to provide a common reference point for the strength of the agents generated.

Our approach to agent evaluation in Connect-4 is as follows: When TDL training is finished, TDL (Yellow) and Minimax (Red) play a tournament of 50 games. (Since Minimax will always win playing Yellow, we consider only games with Minimax playing Red.) The ideal TDL agent is expected to win every game. If both agents act fully deterministically, each game would be identical. We introduce a source of randomness without sacrificing any agent's strength as follows: If Minimax has several optimal moves at its disposal, it chooses one of them randomly. The TDL agent gets a score of 1 for a win, 0.5 for a draw and 0 for a loss. The overall success rate $S_{TDL} \in [0.0, 1.0]$ is the mean of the 50 individual scores. A perfect TDL agent receives a success rate of 1.0.

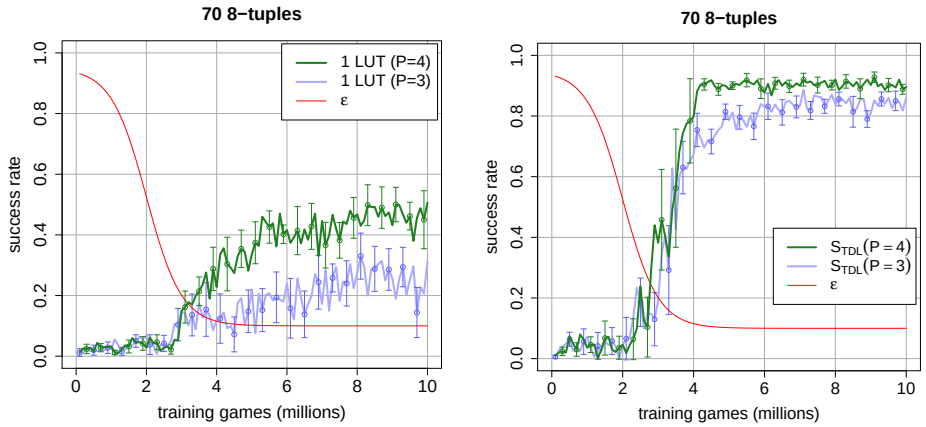
3 Experimental Setup

We established a software framework in Java for conducting learning experiments (see Fig. 1) on the basis of the elements described above: TDL agent with n-tuples, Connect-4, Minimax, agent evaluation and visual inspection capabilities. Each n-tuple is allowed to have one or two LUTs (see Sec. 4).[1]

During training and play, the TDL agent does not use any game-tree search (0-ply look-ahead), instead it just inspects the board positions of its legal moves and chooses the best one. Initially, we tested different n-tuple architectures and decided to conduct all our experiments with 70 n-tuples of length 8 afterwards. If not stated otherwise, the learning rate α followed an exponential decay scheme with $\alpha_{init} = 0.01$ and $\alpha_{final} = 0.001$. Each TDL agent was initialized with weights = 0 and trained during 10 million games of self-play and evaluated every 100,000 games. Table 1 shows the computation time needed on a Q9550 quad-core PC (2.83 GHz on each of the 4 cores) for 10 TDL training experiments.

If we have 70 n-tuples of length 8 with 2 LUTs each, the number of LUT-entries in (P=4)-encoding is $2 \cdot 70 \cdot 4^8 = 9.175 \cdot 10^6$. Note that not every state

[1] The code is available from the authors on request.



(a) No success with 1 LUT for each n-tuple (b) 2 LUTS, one for each player

Fig. 2. Success rate S_{TDL} of the n-tuple-based TDL agent playing Connect-4

corresponding to a LUT-entry is a realizable state during Connect-4 games: If a column's lower cell is 0 or 3, all cells above must be 0. Combinations like (0,1), (0,1,1,2), ... (from bottom to top) are not possible. Thus we expect a large number of weights to stay at zero since they are never updated during training. Tab. 1 confirms this for an example n-tuple set: After training, the number of non-zero weights is only a fraction of the total number of weights.[2]

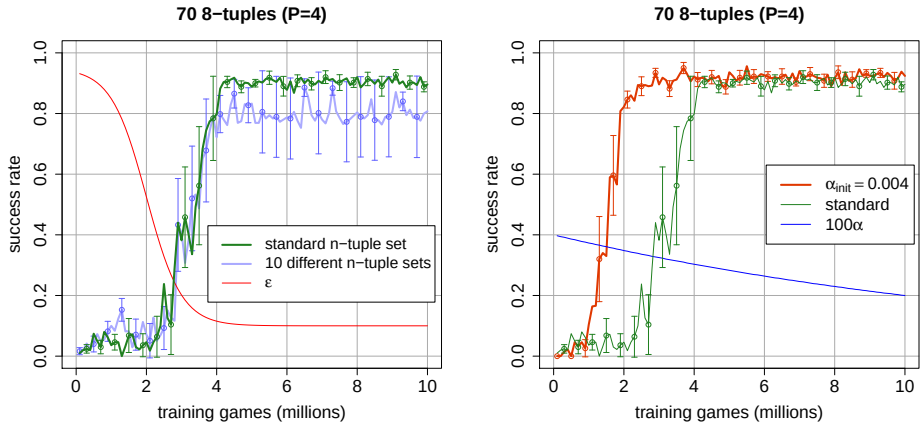
A final important ingredient is the exploration rate ϵ: Connect-4 is a deterministic game and deterministic players would always conduct the same game during self-play. Although a TDL agent might slowly change during learning and thus behaves not fully deterministically, it is nearly deterministic and learning would become ineffective. Therefore, a source of randomness is introduced to aid the exploration of the game tree: With probability ϵ the TDL agent chooses its next move randomly. After such a random move, *no* weight update (Eq. (4)) is performed. The parameter ϵ follows a sigmoidal decay scheme, usually with $\epsilon_{init} = 0.95$ and $\epsilon_{final} = 0.1$.

4 Results

First results, cf. Fig. 2(a), were rather discouraging: The TDL agent rarely won against Minimax, even utilizing different n-tuple creation schemes (random walk, random snake, random sample) and both types of position encoding, i.e. P=3 and P=4. The results show the mean from 10 experiments and even the best experiment has only a mediocre success rate $S_{TDL} < 58\%$.

Two LUTs The major breakthrough was achieved when we changed the n-tuple architecture from 1 LUT per n-tuple to 2 LUTs, one for each player. In contrast to Othello, a certain position in a subpart of the board can have a rather

[2] The number of non-zero weights usually amounts to 98-100% of the realizable states.



(a) 10 different n-tuple sets (b) Improved learning rate decay scheme

Fig. 3. Further results with TDL agent playing Connect-4

different value whether it is Yellow's turn or Red's turn. Therefore, it is advisable to learn both concepts separately. As the light blue curve in Fig. 2(b) shows, we, for the first time, receive an 80% success rate. The results are comparatively stable: If we repeat the experiment ten times with the same n-tuple set but different random moves, we get very similar results, cf. Fig. 2(b). The error bars represent the standard deviation from the ten experiments.

(P=4)-Encoding The darker green curve in Fig. 2(b) shows the beneficial effects of (P=4)-encoding (cf. Sec. 2.2): Compared to the (P=3)-encoding we reach a strong-playing agent much faster (the 80% success rate line is crossed after 4 million games instead of 5 million games), the mean success rate between 6 and 10 million training games is 6% higher (90% instead of 84%), and the variance is lower. We refer to the darker green curve as "standard" in the following experiments.

Different N-tuple Sets The n-tuple creation process is completely by chance, no game specific considerations are taken into account. We tested whether different randomly selected sets would produce a large variance. The results with 10 different sets are shown in Fig. 3(a): The light blue curve is considerably lower (10% in the range beyond $6 \cdot 10^6$ games) and has a larger variance.

When analyzing this result, we found that half of the variance can be attributed to *one* 'bad' n-tuple set, which completely failed on the task (success rate < 0.1). All others sets were very similar to the standard n-tuple set (approx. 4.5% lower in the range beyond $6 \cdot 10^6$ games). The results are consistent: A 'bad' n-tuple set will always be bad when repeating the training with other random moves, a 'good' set will always be good.

Learning Rate Tuning We observed that an α-decay scheme with $\alpha_{final} \geq 0.004$ does not learn anything. A possible reason is that, due to conflicting signals,

individual weights oscillate up and down without the ability to average over different situations. This is often the case in TDL game learning. The sharp onset in success rate in Fig. 2(b) and Fig. 3(a) is with 4 million training games exactly at the point in training where α drops below 0.004. It seems natural to change the α-decay scheme from $[\alpha_{init}, \alpha_{final}] = [0.01, 0.001]$ (standard) to $[\alpha_{init}, \alpha_{final}] = [0.004, 0.002]$. Having implemented this, we observe a much faster training (the 80% success rate line is crossed after 2 instead of 4 million games) and a slight increase in final performance (92% instead of 90%), cf. Fig. 3(b).

We also tested variations in parameter ϵ (exploration decay scheme), but this does not seem to change the results very much. However, a thorough tuning was not performed yet.

5 Conclusion

We summarize by answering our research questions as stated in Sec. 1.2:

Q1 The n-tuple approach has proven to be very successful for the game of Connect-4. For the first time, we generated a strong-playing agent trained just by self-play, i. e. it had no access to teaching information other than the mere rules of the game. The shapes of the n-tuples were not designed by the programmer, instead the system selected them by random walk.

Q2 Lucas found that 1250 training games were enough to produce a strong-playing TDL agent applying the n-tuple approach to Othello [8]. We cannot confirm this for Connect-4. So far we need 2–4 million training games to produce strong-playing Connect-4 agents. One possible reason, among others, is that Othello has an 8-fold symmetry, while it is only a 2-fold symmetry in Connect-4. Thus, it takes longer to visit all branches of the game tree. It is an open question, whether more or better n-tuples will shorten the training time.

Q3 Among the ingredients crucial for the success of learning there is first of all the architecture $\{$*n-tuple + TDL*$\}$ itself: The game Connect-4, despite the fact that it is heuristically solved, seems somewhat hard to code for learning architectures. To the knowledge of the authors, there is no other result in literature, where a learning agent, just trained from self-play, was able to consistently win Connect-4 when playing as starting player against Minimax. Given this architecture, some other ingredients are essential to get the n-tuple approach to work in Connect-4:

- *Two* LUTs per n-tuple, one for each player. Using only one LUT, conflicting signals hinder the TDL architecture to learn.
- It is essential to select the right decay scheme for the learning rate α: With too large α values, the agent fails to learn anything, while a too small α slows down the learning process.

Future work Given the large impact seen in the n-tuple creation process, it is natural to ask whether learning is depending on the characteristics of the n-tuple sets (number, size, and shape of n-tuples). An interesting area of future work is the question whether it is possible to *evolve* n-tuple sets by keeping successful n-tuples in a population and removing bad or redundant ones.

We presented the n-tuple approach to Connect-4 in conjunction with TD learning. It might as well be interesting to compare it with other learning strategies such as co-evolutionary learning (CEL) [7].

We conclude by emphasizing the impressiveness that such a simple and general architecture is able to learn near-perfect decision-making in a complex surrounding *completely from self-play* without a teacher. Here, 'simple' means that no domain-specific knowledge about the tactics of the game is coded, neither explicitly nor implicitly. It is our impression that the mighty feature space, implicitly induced by the large LUTs of the n-tuple approach, plays a large role here.

References

1. V. Allis. A knowledge-based approach of Connect-4. The game is solved: White wins. Master's thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, 1988.
2. W. W. Bledsoe and I. Browning. Pattern recognition and reading by machine. In *Proc. Eastern Joint Computer Conference*, pages 225–232, New York, 1959.
3. D. Curran and C. O'Riordan. Evolving Connect-4 playing neural networks using cultural learning. NUIG-IT-081204, National University of Ireland, Galway, 2004.
4. S. Edelkamp and P. Kissmann. Symbolic classication of general two-player games. Technical report, Technische Universität Dortmund, 2008.
5. W. Konen and T. Bartz-Beielstein. Reinforcement learning: Insights from interesting failures in parameter selection. In G. Rudolph, editor, *Proc. Parallel Problem Solving From Nature (PPSN'2008)*, pages 478–487. Springer, Berlin, 2008.
6. W. Konen and T. Bartz-Beielstein. Reinforcement learning for games: failures and successes – CMA-ES and TDL in comparison. In *Proc. GECCO'2009, Montreal*, pages 2641–2648. ACM, New York, 2009.
7. K. Krawiec and M. G. Szubert. Learning n-tuple networks for Othello by coevolutionary gradient search. In *Proc. GECCO'2011, Dublin*, pages 355–362. ACM, New York, 2011.
8. S. M. Lucas. Learning to play Othello with n-tuple systems. *Australian Journal of Intelligent Information Processing*, 4:1–20, 2008.
9. A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM journal of Research and Development*, 3(3):210–229, 1959.
10. M. Schneider and J. Garcia Rosa. Neural Connect-4 - a connectionist approach. In *Proc. VII. Brazilian Symposium on Neural Networks*, pages 236–241, 2002.
11. P. Sommerlund. Artificial neural nets applied to strategic games. Unpublished, last access: 05.06.12. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.4690, 1996.
12. M. Stenmark. Synthesizing board evaluation functions for Connect-4 using machine learning techniques. Master's thesis, Østfold University College, Norway, 2005.
13. R. S. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst, MA, 1984.
14. R. S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44, 1988.
15. G. Tesauro. TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.
16. M. Thill. Using n-tuple systems with TD learning for strategic board games (in German). CIOP Report 01/12, Cologne University of Applied Science, 2012.