Final Adaptation Reinforcement Learning for N-Player Games [*]

Wolfgang Konen[0000−0002−1343−4209] and
Samineh Bagheri[0000−0003−2488−8000]

Computer Science Institute, TH Köln, University of Applied Sciences, Germany
{wolfgang.konen,samineh.bagheri}@th-koeln.de

Abstract. This paper covers n-tuple-based reinforcement learning (RL) algorithms for games. We present new algorithms for TD-, SARSA- and Q-learning which work seamlessly on various games with arbitrary number of players. This is achieved by taking a player-centered view where each player propagates his/her rewards back to previous rounds. We add a new element called Final Adaptation RL (FARL) to all these algorithms. Our main contribution is that FARL is a vitally important ingredient to achieve success with the player-centered view in various games. We report results on seven board games with 1, 2 and 3 players, including Othello, ConnectFour and Hex. In most cases it is found that FARL is important to learn a near-perfect playing strategy. All algorithms are available in the GBG framework on GitHub.

Keywords: Reinforcement learning · TD-learning · SARSA · game learning · N-player games · n-tuples

1 Introduction

1.1 Motivation

It is desirable to have a better understanding of the principles how computers can learn strategic decision making. Games are an interesting test bed and reinforcement learning (RL) is a general paradigm for strategic decision making. It is however not easy to devise algorithms which work seamlessly on a large variety of games (different rules, goals and game boards, different number of players and so on). It is the hope that finding such algorithms and understanding which elements in them are important helps to better understand the principles of learning and strategic decision making.

Learning how to play games with neural-network-based RL agents can be seen as a complex optimization task. It is the goal to find the right weights such that the neural network outputs the optimal policy for all possible game states or a near-optimal policy that minimizes the expected error. The state space in board games is usually discrete and in most cases too large to be searched exhaustively.

---

[*] This work is an extended version of a paper submitted to *BIOMA '2020, Bruxelles.*

These aspects pose challenges to the optimizer which has to generalize well to unseen states and has to avoid overfitting.

In this paper we describe in detail a new class of n-tuple-based RL algorithms (TD-, SARSA- and Q-learning). N-tuples were introduced by Lucas [12] to the field of game learning. These new learning algorithms extend the work described in [1,8,18] and serve the purpose to be usable for a large variety of games. More specifically we deal here with discrete-time, discrete-action, one-player-at-a-time games. This includes board games and card games with $N = 1, 2, \ldots$ players. Games may be deterministic or nondeterministic.

N-tuple networks are shown to work well in a variety of games, (e.g. in ConnectFour [1,18], Othello [12], EinStein würfelt nicht [3], 2048 [17], SZ-Tetris [7] etc.) but the algorithms described here are not tied to them. Any other function approximation network (deep neural network or other) could be used as well.

All algorithms presented here are implemented in the General Board Game (GBG) learning and playing framework [9,10] and are applied to several games. The variety of games makes the RL algorithms a bit more complex than the basic RL algorithms. This paper describes the algorithm as simple as possible, yet as detailed as necessary to be precise and to follow the implementation in GBG's source code, which is available on GitHub[1].

A work related to GBG [9,10] is the general game systems Ludii [13]. Ludii is an efficient general game system based on ludeme library implemented in Java, allowing to play as well as to generate a large variety of strategy games. Currently all AI agents implemented in Ludii are tree-based agents (MCTS variants or Alpha-Beta). GBG on the other hand offers the possibility to train RL-based algorithms on several games.

The main contributions of this paper are as follows: (i) It presents a unifying view for RL algorithms applicable to different games with different number of players; (ii) it demonstrates that a new element, named Final Adaptation RL (FARL), is vital for having success with this new unifying view; (iii) it incorporates several other elements (afterstates, n-tuples, eligibility with horizon, temporal coherence) that are useful for all games. To the best of our knowledge, this is the first time that these elements are brought together in a comprehensive form for game-learning algorithms with arbitrary number N of players. In addition, we point out and demonstrate that SARSA- and Q-learning have some disadvantages in game learning as compared to TD-learning.

1.2   Algorithm Overview

The most important task of a game-playing agent is to propose, given a game state s_t, a good next action a_t from the set of available actions in s_t. TD-learning uses the value function $V(s_t)$ which is the expected sum of future rewards when being in state s_t. Similarly, Q- and SARSA-learning (**S**tate-**A**ction-**R**eward-**S**tate-**A**ction) use the Q-function $Q(s_t, a_t)$ which is the expected sum of

---

[1] https://github.com/WolfgangKonen/GBG

future rewards when taking action a_t in state s_t. The similarities and differences between these variants are well explained in [14,16].

It is the task of the agents to learn the value function $V(s)$ (TD) or the Q-function $Q(s,a)$ (SARSA, Q) from experience (interaction with the environment). In order to do so, they usually perform multiple self-play training episodes, until a certain training budget is exhausted or a certain game-playing strength is reached.

The nomenclature and algorithmic description follows as closely as possible the descriptions given in [6,17]. But these algorithms are for the special case of the 1-player game 2048. Since we want to use the TD-n-tuple algorithm for a broader class of games, we present in this paper a unified TD-update scheme inspired by [14] which works for 1-, 2-, ..., N-player games.

Our new RL-algorithm is partly inspired by [6,14] and partly from our own experience with RL-n-tuple training. The key elements of the new RL-logic – as opposed to our previous RL-algorithms [1,8] – are:

- New afterstate logic [6], see Sec. 2.2.
- Generalization to N-player games with arbitrary N [14], see Sec. 2.4.
- Final adaptation RL (FARL) for all players, see Sec. 2.7.
- Eligibility method with horizon based on [6], see Sec. 2.3.

The following key elements are also part of our algorithms, but only briefly explained in the appendix of this technical report

- Weight-individual learning rates via temporal coherence learning (TCL) [1,2], see Appendix A.5.
- Faster learning through symmetries [12].
- Modified n-tuple update rule with global learning rate scaling factor. see Appendix A.3.
- Inhibition of multiple changes of the same weight in one update step, see Appendix A.3 and A.4.

This technical report extends the BIOMA'2020 paper insofar that it covers SARSA as well, that it covers the aforementioned algorithmic details in Appendix A.3–A.5, and that it has in Appendix A.6 all the parameter settings for the experiments shown in Tab. 1.

2 Algorithms and Methods

2.1 N-Tuple Systems

N-tuple systems coupled with TD were first applied to game learning by Lucas in 2008 [12], although n-tuples were introduced already in 1959 for character recognition purposes. The remarkable success of n-tuples in learning to play Othello [12] motivated other authors to benefit from this approach for a number of other games. The main goal of n-tuple systems is to map a highly non-linear function in a low dimensional space to a high dimensional space where it is

easier to separate 'good' and 'bad' regions. This can be compared to kernel trick in SVM. An n-tuple is defined as a sequence of n cells of the board. Each cell can have m values representing the possible states of that cell. Therefore, every n-tuple will have a (possibly large) look-up table indexed in form of an n-digit number in base m. An n-tuple system contains multiple n-tuples.

2.2 Afterstate Logic

For nondeterministic games, Jaskowski et al. [6,17] describe a clever mechanism to reduce the complexity of the value function $V(s)$.
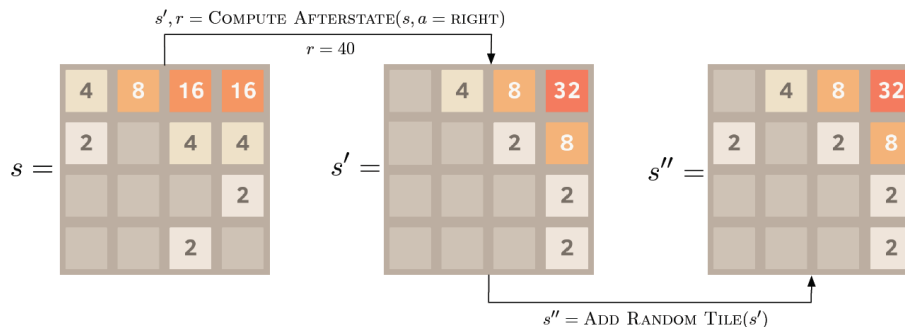

Fig. 1. For nondeterministic games it is better to split a state transition from s to s'' in a deterministic part, resulting in **afterstate** s', and a random part resulting in **next state** s'' (from [17]).

Consider a game like 2048 (Fig. 1): An exemplary action is to move all tiles to the right, this will cause the environment to merge adjacent same-value tiles into one single tile twice as big. This is the deterministic part of the action and the resulting state is called the **afterstate** s'. The second part of the action *move-right* is that the environment adds a random tile 2 or 4 to one of the empty tiles. This results in the **next state** s''.

The naive approach for learning the value function would be to observe the next state s'' and learn $V(s'')$. But this has the burden of increased complexity: Given a state-action pair (s, a) there is only *one* afterstate, but $2n$ possible next states s'', where n is the number of empty tiles in afterstate s'.[2] This makes it much harder to learn the value of an action a in state s. And indeed, it is not the specific value of $V(s'')$ which is the value of action a, but it is the expectation value $\langle V(s'') \rangle$ over *all* possible next states s''.

---

[2] In the example of Fig. 1 we have $n = 9$ empty tiles in afterstate s', thus there are $2n = 18$ possible next states s''. The factor 2 arises because the environment can place one of the two random tiles 2 or 4 in any empty tile.

It is much more clever to learn the value $V(s')$ of an afterstate. This reduces the complexity by a factor of $2\bar{n}$, where $\bar{n}$ is the average number of empty tiles. It helps the agent to generalize better in all phases of TD-learning.

For deterministic games there is no random part: afterstate s' and next state s'' are the same. However, afterstates are also beneficial for deterministic games: For positional games (like TicTacToe, ConnectFour, Hex, ...) the value of taking action a in state s depends only on the resulting afterstate s'. Several state-action pairs might lead to the same afterstate, and it often reduces the complexity of game learning if we learn the the mapping from afterstates to game values (as we do in TD-learning, Sec. 2.5). This is opposed to learning the mapping from state-action pairs to values (as we do in SARSA- and Q-learning, Sec. 2.6).

2.3 Eligibility Method

Instead of Sutton's eligibility traces [16] we use in this work Jaskowski's eligibility method [6]. This method is efficiently computable even in the case of long RL episodes and it can be made equivalent to eligibility traces in the case of short episodes. For details the reader is referred to Appendix A.3 or to [6].

2.4 N Players

We want to propose a general TD(λ) n-tuple algorithm which is applicable not only to 1- and 2-player games but to arbitrary N-player games.

The key difference to the TD-learning variants described in earlier work [1,8] is that there each state was connected with the next state in the episode. This required different concepts for TD-learning, depending on whether we had a 1- or a 2-player game (minimization or maximization). Furthermore it has a severe problem for N-player games with $N > 2$: We usually do not know the game value for all other players in intermediate states, but we would need them for the algorithms in [1,8]. In contrast, Ree and Wiering [14] describe an approach where each player has a value function only for *his/her* states s_t or state-action-pairs (s_t, a_t). The actions of the opponents are subsumed in the reaction from the environment. That is, if s_t is the state for player p_t at time t, then s_{t+1} is the next state of the *same* player p_t on which (s)he has to act. This has the great advantage that there is no need to translate the value of a state for player p_t to the value for other players – we take always the perspective of the same player when calculating temporal differences.[3]

This holds for TD-, SARSA- and Q-learning. In the next section we describe the application of these ideas to TD-learning, which will result in the (new) TD-FARL n-tuple algorithm valid for all N-player games.

2.5 TD Learning for N Players

We set up a TD-learning algorithm connecting moves to the last move of the **same** player. This is done in Algorithm 1 (TDFROMEPISODE). Algorithm 1

---

[3] If the player uses a neural network for function approximation it may or may not be that the opponents also use the same network [14].

Algorithm 1 TDFROMEPISODE: Perform one episode of TD-learning, starting from state s_0. States s'_{t-1}, s_t, s'_t and actions a_t are for **one** specific player p_t. r_t is the delta reward for p_t when taking action a_t in state s_t. A_t is the set of actions available in state s_t.

1: **function** TDFROMEPISODE(s_0)
2:     $t \leftarrow 0$
3:     **repeat**
4:         Choose for player p_t action $a_t \in A_t$ from s_t using policy derived from V
5:                 ▷ e.g. ϵ-greedy: with probability ϵ random, with prob. $1 - \epsilon$ using V
6:         Take action a_t and observe reward r_t, afterstate s'_t and next state s''.
7:         $V^{new}(s'_{t-1}) = r_t + \gamma V(s'_t)$ ▷ target value for p_t's previous afterstate
8:         Use NN to get the current value of previous afterstate: $V(s'_{t-1})$
9:         Adapt NN by backpropagating error $\delta = V^{new}(s'_{t-1}) - V(s_{t-1})$
10:         $t \leftarrow t + 1$
11:         $s_t \leftarrow s''$
12:     **until** s'' is terminal

---

shows the TD-learning algorithm in compact form. It thus makes the general principle clear. But it has the disadvantage that it obscures one important detail: What is shown within the while loop is what has to be done by player p_t in state s_t. After completing this, we do however *not* move to the next state s_{t+1} of the same player p_t (one round away), but we let the environment act, get a new state s''_t for the **next** player, and then this next player does *his/her* pass through the while loop.

To make these details more clear, we write the algorithm down in a form where the pseudocode is closer to the GBG implementation. This is done in Algorithm 2 (TD-FARL-EPISODE). Some remarks on Algorithm 2:

- Now the sequence of states $s_0, s_1, ..., s_f$ is really the sequence of consecutive moves in an episode. The players usually vary in cyclic order, $0, 1, ..., N - 1, 0, 1, ...$, but other turn sequences are possible as well.
- In each state the connection to the last afterstate of the same player p is made via $\boldsymbol{s}_{last}[p]$. Thus the update step is equivalent to Algorithm 1.
- In contrast to Algorithm 1, this algorithm has the final adaptation step FARL (function FINALADAPTAGENTS) included. FARL is covered in more detail in Sec. 2.7.

Algorithm 2 is simpler and at the same time more general than our previous TD-algorithms [1,8] for several reasons:

1. Each player has its own value function V and each player seeks actions that **maximize** this V. This is because each V has in its targets the rewards from the perspective of the acting player. So there is no need to set up complicated cases distinguishing between minimization and maximization as it was [1,8].
2. The same algorithm is viable for **arbitrary** number of players.

Algorithm 2 TD-FARL-EPISODE: Perform one episode of TD-learning, starting from state s_0. Similar to Algorithm 1, but with Final-Adaptation RL (FARL). We connect afterstate s' via player p_t with the previous afterstate $\boldsymbol{s}_{last}[p_t]$ of this player. Note that $\boldsymbol{s}_{last}$ and $\boldsymbol{r}$ are vectors of length N.

1: **function** TD-FARL-EPISODE(s_0)
2:     $t \leftarrow 0$;
3:     $\boldsymbol{s}_{last}[p] \leftarrow$ null $\forall$ player $p = 0, \ldots, N - 1$ ▷ last afterstates
4:     **repeat**
5:         $p_t =$ player to move in state s_t
6:         Choose action a_t from s_t using policy derived from V ▷ e.g. ϵ-greedy
7:         $(\boldsymbol{r}, s', s'') \leftarrow$ MAKEACTION(s_t, a_t) ▷ s': afterstate (after taking a_t)
8:                          ▷ $\boldsymbol{r}$ is the delta reward tuple from the perspective of **all** players p
9:         ADAPTAGENTV$(\boldsymbol{s}_{last}[p_t], \boldsymbol{r}[p_t], s')$
10:         $\boldsymbol{s}_{last}[p_t] \leftarrow s'$ ▷ the afterstate generated by p_t when taking action a_t
11:         $t \leftarrow t + 1$
12:         $s_t \leftarrow s''$
13:     **until** (s'' is terminal)
14:     FINALADAPTAGENTS$(p_t, \boldsymbol{r}, s')$ ▷ use final reward tuple to adapt all agents
15:
16:                              ▷ Update the value function (based on NN) for player p_t
17: **function** ADAPTAGENTV$(\boldsymbol{s}_{last}[p_t], r', s')$
18:     **if** $(\boldsymbol{s}_{last}[p_t] \neq$ null) **then**   ▷ Adapt $V(\boldsymbol{s}_{last}[p_t])$ towards target T
19:         Target $T = r' + \gamma V(s')$ for afterstate $\boldsymbol{s}_{last}[p_t]$
20:         Use NN to get $V(\boldsymbol{s}_{last}[p_t])$
21:         Adapt NN by backpropagating error $\delta = T - V(\boldsymbol{s}_{last}[p_t])$
22:
23:                              ▷ Terminal update of value function for all players
24: **function** FINALADAPTAGENTS$(p_t, \boldsymbol{r}, s')$
25:     **for** $(p = 0, \ldots, N - 1,$ but $p \neq p_t)$ **do**
26:         **if** $(\boldsymbol{s}_{last}[p] \neq$ null) **then**   ▷ Adapt $V(\boldsymbol{s}_{last}[p])$ towards target $\boldsymbol{r}[p]$
27:             Use NN to get $V(\boldsymbol{s}_{last}[p])$
28:             Adapt NN by backpropagating error $\delta = \boldsymbol{r}[p] - V(\boldsymbol{s}_{last}[p])$
29:                              ▷ Adapt $V(s') \to 0$ (s': terminal afterstate of player p_t)
30:     Use NN to get $V(s')$
31:     Adapt NN by backpropagating error $\delta = 0 - V(s')$

---

3. There is no (or less) unwanted crosstalk because of too frequent updates (as it was the case for some variants in [1,8]).[4]

4. Since states are connected with states one round (and not one move) earlier, positive or negative rewards propagate back faster.

These advantages apply as well to SARSA- and Q-learning which we will cover next.

2.6 SARSA- and Q-Learning for N Players

SARSA- and Q-learning have a Q-function instead of the value function in TD-learning. Again, each player has a Q-function only for *his/her* state-action-pairs (s_t, a_t). The actions of the opponents are subsumed in the reaction from the environment. That is, if s_t is the state for player p_t at time t, then s_{t+1} is the next state of the *same* player p_t on which (s)he has to act. For the detailed algorithms (e.g. Algorithm 5, SARSA-FARL-EPISODE) the reader is referred to Appendix A.2.

Another key difference is the form of the neural network, as shown in Fig. 2. In TD-learning the network learns $V(s)$ and thus has a single output. It is asked for all afterstates s' arising from available actions as $s' = A(s, a)$ and selects that afterstate that produces the highest $V(s')$. On the other hand, the neural network in Q-learning or SARSA approximates $Q(s, a)$: it has s (or features derived from s) as input, but it has **multiple outputs**, one for each possible action a. Given a certain state s it produces a vector of outputs $Q(s, a)$, one element for each a. SARSA- or Q-learning selects that action a with the highest output activation.
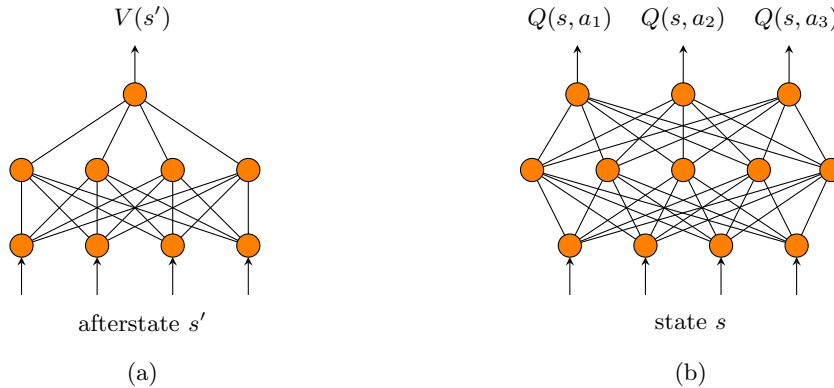

Fig. 2. Neural networks in (a) TD-learning, (b) SARSA- and Q-learning.

---

[4] With crosstalk we mean the effect that the update of the value function for one state has detrimental effects on the learned values for other states.

SARSA- and Q-learning share with TD-learning the same advantages as mentioned at the end of Sec. 2.5. There are however some disadvantages of SARSA- and Q-learning in games:

– If N_a is the number of possible actions, it needs N_a as many weights in the last layer. Since in n-tuple learning there is only one layer, these are N_a as many weights in total. Consequently, the agents are bigger (in memory and on disk) by a factor N_a. This is a considerable increase for larger Hex boards (e.g. 9×9 Hex $\rightarrow N_a = 81$) and other games with high branching factor.
– SARSA and Q-learning cannot exploit the benefits of afterstates: Several state-action pairs (s, a) may lead to the same afterstate s'. Usually the value of an action is fully determined by the resulting afterstate. In such cases, TD-learning stores the value for that afterstate only once, while SARSA- and Q-learning store it multiple times.[5]

These disadvantages of SARSA- and Q-learning are not present in the new TD-learning algorithm 2.

2.7   Final Adaptation RL (FARL)

Once an episode terminates, we have a delta reward tuple for all players. A drawback of the plain TD- and SARSA-algorithms is that only the current player (who generated the terminal state) uses this information to perform an update step. But the other players can also learn from their (usually negative) rewards. This is what the first part of FINALADAPTAGENTS (lines 26-28 of Algorithm 2) is for: Collect for each player *his* terminal delta reward and use this as target for a final update step where the value of the player's state one round earlier is adapted towards this target.[6]

One might ask whether it is not a contradiction to Sec. 2.4 where we stated that the value for other players is not known for $N > 2$. This is not a contradiction: Although intermediate *values* are usual not known for all players, the final *reward* of a game episode – at least for all games we know of – *is available* for all players. It is thus a good strategy to use this information for all players.

Second part of FINALADAPTAGENTS, lines 29-31: A terminal state is by definition a state where no future rewards are expected. Therefore the value of that state should be zero. However, crosstalk in the network due to the adaptation of other states may lead to non-zero values for terminal states. Jaskowski [6] proposes to make an adaptation step towards target 0 for all terminal states.

---

[5] How many times? – If an afterstate s' contains x pieces of the player who just created s', then there are x preceding states leading with the appropriate action to that same afterstate s'. For TicTacToe $x \leq 4$, but for 9×9 Hex $x \leq 40$ (!).

[6] Why does the target have only the delta reward $\boldsymbol{r}[p]$ and does not need the value function $V(s')$? – Because the value function for a terminal s' is always 0 (no future rewards are expected).

3   Results

We show results of our algorithms on two games. In preliminary experiments we tested various settings for parameters, namely the learning rate α, the random move rate ϵ and the eligibility rate λ. We selected for TicTacToe parameter α linearly decreasing from 1.0 to 0.5 and the n-tuple system consisted just of one 9-tuple. For ConnectFour we used $\alpha = 3.7$ and an n-tuple system consisting of initially randomly chosen but then fixed 70 8-tuples. For both games we had ϵ linearly decreasing from 0.1 to 0.0, $\lambda = 0.0$ and we used the TCL scheme as described in [1] and Appendix A.5. Note that due to TCL the effective learning rate adopted by most weights can be far smaller than α.

3.1   TicTacToe

Fig. 3 shows learning curves of TD- and SARSA-learning. The red curves show the full Algorithms 2 (TD-FARL-Episode) and 5 (Sarsa-FARL-Episode). Algorithm 2 is slightly quicker in convergence, but both algorithms will eventually master the simple game TicTacToe. The blue curves show the results when we switch off FinalAdaptAgents: The decrease in performance is severe: drastic for TD and strong for SARSA.
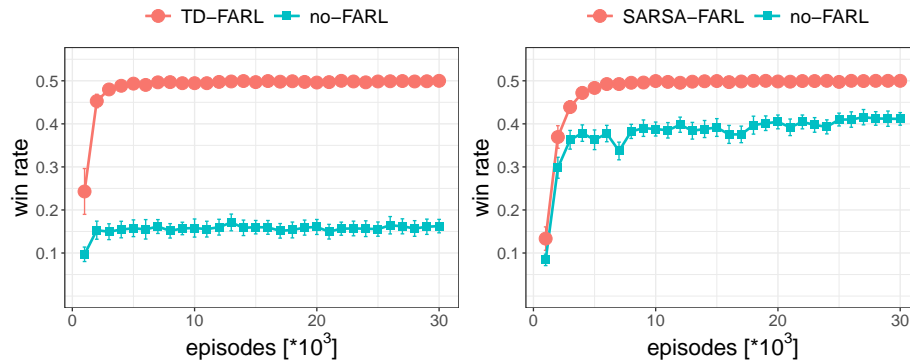

Fig. 3. Different versions of TD-learning (left) and SARSA-learning (right) on TicTac-Toe. Each agent is evaluated by playing games from different start positions in both roles, 1st and 2nd player, against the perfect-playing Max-N agent. The best achievable result is 0.0, because Max-N will win at least in one of the both roles. Shown is the average over 25 training runs.

3.2   ConnectFour

Fig. 4 shows learning curves of our TD-FARL agent on the non-trivial game ConnectFour. Two modes of evaluation are shown: The red curves evaluate

against opponent AlphaBeta (AB), the blue curves against opponent AlphaBeta-Distant-Losses (AB-DL). The AlphaBeta algorithm extends the Minimax algorithm by efficiently pruning the search tree. Thill et al. [18] were able to implement AlphaBeta for ConnectFour in such a way that it plays perfect in situations where it can win. AB and AB-DL differ in the way they react on losing states: While AB just takes a random move, AB-DL searches for the move which postpones the loss as far (distant) as possible. It is tougher to win against AB-DL since it will punish every wrong move. The final results for our TD-FARL agent are however very satisfying: 49.5% win rate against AB, 46.5% win rate against AB-DL. It is worth noting that two perfect-playing opponents (AB and AB-DL) are not necessarily equally strong.
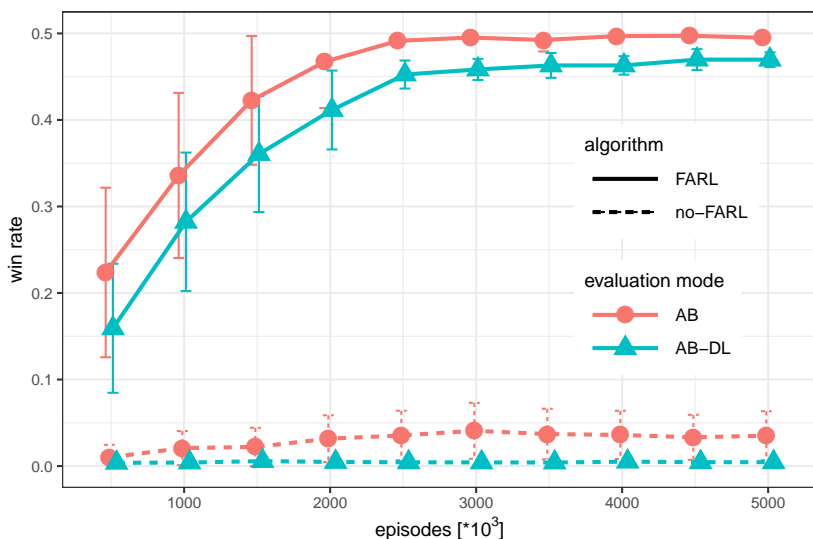

Fig. 4. TD-learning on ConnectFour. During training, agent TD-FARL is evaluated against the perfect-playing agents AlphaBeta (AB) and AlphaBeta-with-distant-losses (AB-DL). Both agents play in both roles (first or second). Since ConnectFour is a theoretical win for the starting player, the ideal win rates against AB and AB-DL are 0.5. The solid lines show the mean win rates from 10 training runs with FARL. The dashed curves *no-FARL* show the results when FARL is turned off. Error bars depict the standard deviation of the mean.

It is a remarkable success that TD-FARL learns only from training by self-play to defeat the perfect-playing AlphaBeta agents in 49%/46% of the cases. Remember that TD-FARL has never seen AlphaBeta before during training. The result is similar to our previous work [1]. But the difference is that the new algorithm can be applied without any change to other games with any N.

Table 1. Results for Algorithm 2 (TD-FARL-Episode) on various games. In Nim(3P) hxs, there are initially h heaps with s stones. For each game, 10 training runs with different seeds are performed and the resulting TD agent is evaluated by playing against opponents as indicated in column 3 (two such opponents in the case of Nim3P). Each agent plays all roles. Shown are the TD agent's win rates or scores (rewards): mean from 10 runs plus/minus one standard deviation of the mean.

Game	N	evaluated vs.	win rates or scores		other RL research	
			FARL	no-FARL		
2048	1		$142\,000 \pm 1\,000$	$122\,000 \pm 900$	[6]	$80\,000$
TicTacToe	2	Max-N$_{10}$ [11]	$49\% \pm 5\%$	$18\% \pm 6\%$		
ConnectFour	2	AB [18]	$49.5\% \pm 0.5\%$	$3.5\% \pm 0.1\%$	[4]	$0.0\% \pm 0.0\%$
		AB-DL [18]	$46.5\% \pm 0.5\%$	$0.0\% \pm 0.1\%$		
Hex 6x6	2	MCTS$_{10000}$	$81\% \pm 5\%$	$0.0\% \pm 0.2\%$		
Othello	2	Edax$_{d1}$ [5]	$55\% \pm 1\%$	$53\% \pm 1\%$		
		Bench [14]	$95\% \pm 0.3\%$	$96\% \pm 0.2\%$	[14]	$87.1\% \pm 0.9\%$
Nim 3x5	2	Max-N$_{15}$ [11]	$50\% \pm 1\%$	$12\% \pm 6\%$		
Nim3P 3x5	3	Max-N$_{15}$ [11]	0.33 ± 0.03	0.03 ± 0.01		
		MCTS$_{5000}$	0.78 ± 0.02	0.09 ± 0.02		

There is also a striking failure visible in Fig. 4: If we switch off FinalAdapt-Agents (curves *no-FARL*), we see a complete break-down of the TD agent: It loses nearly all its games. We conclude that the part propagating the final reward of the other player back to the other player's previous state is vitally important.[7] If we analyze the *no-FARL*-agent we find that it has only 0.9% active weights while the good-working TD-FARL agent has 8% active weights. This comes because the other player (that is the one who loses the game since the current player created a winning state) has never the negative reward propagated back to previous states of that other player. Thus the network fails to learn threatening positions and/or precursors of such threatening positions.

3.3   A Variety of Games

In Table 1 we show the results for seven games with varying number of players (1, 2, or 3). While there exist many well-known games for 1 and 2 players, it is not easy to find 3-player games which have a clear winning strategy. Nim3P, the 3-player-variant of the game Nim, is such a game. Each player can take any number of pieces from one heap at his/her turn. The player who takes the last piece loses and gets a reward of 0.0, then the successor is the winner and gets a reward of 1; the predecessor gets a reward of 0.2. This smaller reward helps to break ties in otherwise 'undecided' situations. The goal for each player is to maximize his/her average reward. Nim3P cannot end in a tie.

---

[7] It is really the first part of FARL which is important: We conducted an experiment where we switch off only the second part of FARL and observed only a very slight degradation (1% or less).

All games are learned by exactly the same TD-FARL / no-FARL algorithm. The strength of the resulting agent is evaluated by playing against opponents, where all agents play in all roles. The opponents are in many cases perfect-playing or strong-playing agents. If all agents play perfect, the best possible result for each agent is a win rate of 50% for 2-player games and a score of 0.4 for the game Nim3P (one third of the total reward 1.2 distributed in each episode). Max-N_d is an N-Player tree search with depth d [11], being a perfect player for the games TicTacToe, Nim, Nim3P. For ConnectFour, AB and AB-DL [18] are perfect-playing agents introduced in Section 3.2. Edax$_{d1}$ [5] is a strong Othello program, played here with depth 1. BENCH [14] is a medium-strength Othello agent. MCTS$_a$ is a Monte Carlo Tree Search with a iterations.

As can be seen from Table 1, TD-FARL reaches near-perfect playing strength in most competitions against (near-)perfect opponents and it dominates non-perfect opponents. The most striking feature of Table 1 is its column 'no-FARL': it is in all games much weaker, with one notable exception: In Othello the results for TD-FARL and TD-no-FARL are approximately the same. This is supported by the results from van der Ree and Wiering [14] who had good results on Othello with their no-FARL algorithms. We have no clear answer yet why Othello behaves differently than all other games.

3.4   Comparison with Other RL Research

For some games we compare in Table 1 with other RL approaches from the literature. Jaśkowski [6] achieves for the game 2048 with a similar amount of training episodes and a general-purpose baseline TD agent scores around 80 000. It has to be noted that Jaśkowski with ten times more training episodes and algorithms specifically designed for 2048 reaches much higher scores around 600 000, but here we only want to compare with general-purpose RL approaches.

Dawson [4] introduces a CNN-based and AlphaZero-inspired [15] RL agent named ConnectZero for the game ConnectFour, which can be played online. Although it reaches a good playing strength against MCTS$_{1000}$, it cannot win a single game against our AlphaBeta agent. We performed 10 episodes with ConnectZero starting (which is a theoretical win), but found that instead AlphaBeta playing second won all games. This is in contrast to our TD-FARL which wins nearly all episodes when starting against AlphaBeta.

Finally we compare for the game Othello with the work of van der Ree and Wiering [14]: Their Q-learning agent reaches against BENCH (positional player) a win rate of 87% while their TD-learning agent reaches 72%. Both win rates are a bit lower than our 95%.

3.5   Discussion

Looking at the results for ConnectFour, one might ask the following question: If FARL is so important for RL-based ConnectFour, why could Bagheri et al. [1] learn the game when their algorithm did not have FARL? – The reason is, that both algorithms have different TD-learning schemes: While the algorithm in [1]

propagates the target from the current state back to the previous state (one *move* earlier), our N-player RL propagates the target from the current state back to the previous move of the same player (one *round* earlier). The N-player FARL is more general (it works for arbitrary N). But it has also this consequence: If for example a 2-player game is terminated by a move of player 1, the value of the previous state $s_{last}[p_2]$ of player 2 is never updated. As a consequence, player 2 will never learn to avoid the state preceding its loss. Exactly this is cured, if we activate FARL.

4   Conclusion and Future Work

In summary, we collected evidence that Algorithm 2 (TD-FARL-Episode) produces good results on a variety of games. It has been shown that the new ingredient FARL (the final adaption step) is vital in many games to get these good results.

Compared to [1], TD-FARL has the benefit that it can be applied unchanged to all kind of games whether they have one, two or three players. The algorithm of [1] cannot be applied to games with more than two players.

We see the following lines of direction for future work: (a) More 3-player games. Although Nim3P with a clear winning strategy provided a viable testbed for evaluating our algorithm, taking more 3-player or N-player games into account will help us to investigate how well our introduced methods generalize. (b) Can we better understand why Othello is indifferent to using FARL or no-FARL? Are there more such games? If so, an interesting research question would be whether it is possible to identify common game characteristics that allow to decide whether FARL is important for a game or not.

References

1. Bagheri, S., Thill, M., Koch, P., Konen, W.: Online adaptable learning rates for the game Connect-4. IEEE Transactions on Computational Intelligence and AI in Games **8**(1), 33–42 (2015)
2. Beal, D.F., Smith, M.C.: Temporal coherence and prediction decay in TD learning. In: Dean, T. (ed.) Int. Joint Conf. on Artificial Intelligence (IJCAI). pp. 564–569. Morgan Kaufmann (1999)
3. Chu, Y.R., Chen, Y., Hsueh, C., Wu, I.: An agent for EinStein Würfelt Nicht! using n-tuple networks. In: 2017 Conference on Technologies and Applications of Artificial Intelligence (TAAI). pp. 184–189 (Dec 2017)
4. Dawson, R.: Learning to play Connect-4 with deep reinforcement learning (2020), https://codebox.net/pages/connect4, retrieved August,21,2020
5. Delorme, R.: Edax, version 4.4 (2019), https://github.com/abulmo/edax-reversi, retrieved August,1,2020
6. Jaśkowski, W.: Mastering 2048 with delayed temporal coherence learning, multi-stage weight promotion, redundant encoding, and carousel shaping. IEEE Transactions on Games **10**(1), 3–14 (2018)

7. Jaśkowski, W., Szubert, M., Liskowski, P., Krawiec, K.: High-dimensional function approximation for knowledge-free reinforcement learning: A case study in SZ-Tetris. In: Conf. on Genetic and Evolutionary Computation. pp. 567–573 (2015)
8. Konen, W.: Reinforcement learning for board games: The temporal difference algorithm. Tech. rep., TH Köln (2015), http://www.gm.fh-koeln.de/ciopwebpub/Kone15c.d/TR-TDgame_EN.pdf
9. Konen, W.: General board game playing for education and research in generic AI game learning. In: Perez, D., Mostaghim, S., Lucas, S. (eds.) Conference on Games (London). pp. 1–8 (2019), https://arxiv.org/pdf/1907.06508
10. Konen, W.: The GBG class interface tutorial V2.1: General board game playing and learning. Tech. rep., TH Köln (2020), http://www.gm.fh-koeln.de/ciopwebpub/Konen20a.d/TR-GBG.pdf
11. Korf, R.E.: Multi-player alpha-beta pruning. Artificial Intelligence **48**(1), 99–111 (1991)
12. Lucas, S.M.: Learning to play Othello with n-tuple systems. Australian Journal of Intelligent Information Processing **4**, 1–20 (2008)
13. Piette, É., Soemers, D.J.N.J., Stephenson, M., Sironi, C.F., Winands, M.H.M., Browne, C.: Ludii - the ludemic general game system. CoRR **abs/1905.05013** (2019), http://arxiv.org/abs/1905.05013
14. van der Ree, M., Wiering, M.: Reinforcement learning in the game of Othello: Learning against a fixed opponent and learning from self-play. In: Adaptive Dynamic Programming and Reinforcement Learning (ADPRL). pp. 108–115 (2013)
15. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al.: Mastering the game of Go without human knowledge. Nature **550**(7676), 354–359 (2017)
16. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA (1998)
17. Szubert, M., Jaśkowski, W.: Temporal difference learning of n-tuple networks for the game 2048. In: Computational Intelligence and Games (CIG), 2014 IEEE Conference on. pp. 1–8. IEEE (2014)
18. Thill, M., Bagheri, S., Koch, P., Konen, W.: Temporal difference learning with eligibility traces for the game Connect-4. In: Preuss, M., Rudolph, G. (eds.) International Conf on Computational Intelligence in Games (CIG), Dortmund (2014)

A   Appendix

Table 2. Symbols and variables

symbol	text
s_t	game state at time t
a_t	action taken at time t
A_t	set of actions available in state s_t
p_t	$p_t \in \{0, \ldots, N-1\}$: player to move in state s_t
N	number of players in the game
r_{t+1}	reward delta when arriving with (s_t, a_t) in state s_{t+1}
R_{t+1}	absolute (cumulative) reward in state s_{t+1}
T_{lrnRM}	boolean switch: If set to true, then TD-update shall be done after a random action (see line 2 of Algorithm 7)
$\epsilon \in [0, 1]$	random move rate
$\gamma \in [0, 1]$	discount factor, usually 1.0

A.1   Nomenclature

We list in Table 2 the symbols and variables used in this paper.

A.2   SARSA and Q-Learning: Detailed Algorithms

Algorithm 3 and 4 show Q-learning and SARSA algorithms in their shortest form, close to [14,16], solely from the perspective of one player. s_t and s_{t+1} are one *round* away (same player, next round). These algorithms are quite compact, but again they omit one detail: After a pass through the while loop with state s_t, the next pass is *not* with state s_{t+1} (same player, next round), but with the state s'' resulting from advancing s_t with a_t. Usually, s'' has another player $p'' = (p_t + 1)\%N$.

It is tempting, but dangerous (or plain wrong) to try to work in Algorithm 4 with only *one* 'Choose action a_t' per training cycle: Could we not re-use the a_{t+1} obtained in every pass as the a_t of the next pass through the while loop? - The answer is *No* because after choosing a_{t+1} we change Q, and thus the action obtained from s_{t+1} may be different.

Algorithm 5 shows the SARSA algorithm in a somewhat longer form exposing this and other details. It is closer to the GBG-implementation and it is for the case that we have one NN (the n-tuple network) used by all players and that training proceeds by having all SARSA agents play against themselves. Now s_t and s_{t+1} are one *move* away (next player). We store in $\boldsymbol{s}_{last}[p]$ the last state of the same player p one round earlier and thus make the update step equivalent to Algorithm 4.

There is with FINALADAPTAGENTS one more important difference to algorithm 4: Once an episode is finished (s_{t+1} is terminal), we add one fictious round

Algorithm 3 QLEARNFROMEPISODE: Perform one episode of Q-learning, starting from state s_0. States s_t and actions a_t are for **one** specific player p. r_{t+1} is the delta reward for p when taking action a_t in state s_t. A_t is the set of actions available in state s_t. Additional inputs are: γ (discount factor), and ϵ (random move rate).

1: **function** QLEARNFROMEPISODE(s_0)
2:     $t \leftarrow 0$
3:     **while** true **do**
4:         Choose action $a_t \in A_t$ from s_t using policy derived from Q
5:             ▷ e.g. ϵ-greedy: with probability ϵ random, with prob. $1 - \epsilon$ using Q
6:         Execute a_t and observe r_{t+1}, s_{t+1} ▷ after environment and opponents act
7:         Use NN to get $Q(s_{t+1}, a')$ $\forall a' \in A_{t+1}$
8:         $M \leftarrow \max_{a' \in A_{t+1}} Q(s_{t+1}, a')$ ▷ if s_{t+1} is terminal, set $M = 0$
9:         Target $Q^{new}(s_t, a_t) = r_{t+1} + \gamma M$
10:        Use NN to get $Q(s_t, a_t)$ ▷ only **one** action a'
11:        Adapt NN by backpropagating error $\delta = Q^{new}(s_t, a_t) - Q(s_t, a_t)$
12:                ▷ NN-outputs for all actions other than a_t have their δ set to 0
13:        **if** s_{t+1} is terminal **then break**   ▷ out of while
14:        $t \leftarrow t + 1$

Algorithm 4 SARSAFROMEPISODE: Perform one episode of SARSA-learning, starting from state s_0. States s_t and actions a_t are for **one** specific player p. r_{t+1} is the delta reward for p when taking action a_t in state s_t. A_t is the set of actions available in state s_t. Additional inputs are: γ (discount factor), and ϵ (random move rate).

1: **function** SARSAFROMEPISODE(s_0)
2:     $t \leftarrow 0$
3:     **while** true **do**
4:         Choose action $a_t \in A_t$ from s_t using policy derived from Q
5:             ▷ e.g. ϵ-greedy: with probability ϵ random, with prob. $1 - \epsilon$ using Q
6:         Execute a_t and observe r_{t+1}, s_{t+1} ▷ after environment and opponents act
7:         Choose action $a_{t+1} \in A_{t+1}$ from s_{t+1} using policy derived from Q
8:         $M \leftarrow Q(s_{t+1}, a_{t+1})$ ▷ if s_{t+1} is terminal, set $M = 0$
9:         Target $Q^{new}(s_t, a_t) = r_{t+1} + \gamma M$
10:        Use NN to get $Q(s_t, a_t)$ ▷ only **one** action a_t
11:        Adapt NN by backpropagating error $\delta = Q^{new}(s_t, a_t) - Q(s_t, a_t)$
12:                ▷ NN-outputs for all actions other than a_t have their δ set to 0
13:        **if** s_{t+1} is terminal **then break**   ▷ out of while
14:        $t \leftarrow t + 1$

Algorithm 5 SARSA-FARL-EPISODE: Equivalent to SARSAFROMEPISODE, but the pseudocode is closer to the GBG implementation. s'' is the next state from s_t after taking action a_t, usually with another player p_{next} to move. We connect it via p_{next} with the previous state $\boldsymbol{s}_{last}[p_{next}]$ of this player. Note that $\boldsymbol{s}_{last}, \boldsymbol{a}_{last}$ are vectors of length N.

1: **function** SARSA-FARL-EPISODE(s_0)
2:     $t \leftarrow 0$
3:     Choose action a_0 from s_0 using policy derived from Q ▷ e.g. ϵ-greedy, using NN
4:     $\boldsymbol{s}_{last}[p] \leftarrow (p = p_0)\,?\,s_0 : \text{null} \quad \forall\,\text{player}\,p = 0, \ldots, N-1$
5:     $\boldsymbol{a}_{last}[p] \leftarrow (p = p_0)\,?\,a_0 : \text{null} \quad \forall\,\text{player}\,p = 0, \ldots, N-1$
6:     **repeat**
7:         $(\boldsymbol{r}, s', s'') \leftarrow \text{MAKEACTION}(s_t, a_t)$ ▷ afterstate s' is not used
8:                     ▷ $\boldsymbol{r}$ is the delta reward tuple from the perspective of **all** players p
9:         $p_{next} = $ player to move in next state s''
10:        ADAPTAGENTQ($p_{next}, \boldsymbol{r}[p_{next}], s''$)
11:        Choose action a'' from s'' using policy derived from Q
12:                     ▷ e.g. ϵ-greedy, using NN. Calculate a'' anew after ADAPTAGENTQ(()!)
13:        $t \leftarrow t + 1$
14:        $\boldsymbol{s}_{last}[p_{next}] \leftarrow s_t \leftarrow s''$
15:        $\boldsymbol{a}_{last}[p_{next}] \leftarrow a_t \leftarrow a''$
16:     **until** (s'' is terminal)
17:     FINALADAPTAGENTS($p_{next}, \boldsymbol{r}$)   ▷ use final reward tuple to adapt other agents
18:
19:                     ▷ Update the Q-function (based on NN) for player p to act in s''
20: **function** ADAPTAGENTQ(p, r'', s'')
21:     **if** (s'' is terminal) **then**
22:         $M = 0$
23:     **else**
24:         Choose action a'' from s'' using Q-derived policy ▷ e.g. ϵ-greedy, using NN
25:         $M \leftarrow Q(s'', a'')$
26:     **if** ($\boldsymbol{s}_{last}[p] \neq \text{null}$) **then**   ▷ Adapt Q towards target T
27:         Target $T = r'' + \gamma M$
28:         Use NN to get $Q(\boldsymbol{s}_{last}[p], \boldsymbol{a}_{last}[p])$ ▷ only **one** action
29:         Adapt NN by backpropagating error $\delta = T - Q(\boldsymbol{s}_{last}[p], \boldsymbol{a}_{last}[p])$
30:                 ▷ NN-outputs for all actions other than $\boldsymbol{a}_{last}[p]$ have their δ set to 0
31:
32:                     ▷ Terminal update of Q-functions for all players **other** than p_{next}
33: **function** FINALADAPTAGENTS($p_{next}, \boldsymbol{r}$)
34:     **for** ($p = 0, \ldots, N-1, \text{but}\, p \neq p_{next}$) **do**
35:         **if** ($\boldsymbol{s}_{last}[p] \neq \text{null}$) **then**   ▷ Adapt Q towards target $T = \boldsymbol{r}[p]$
36:             Use NN to get $Q(\boldsymbol{s}_{last}[p], \boldsymbol{a}_{last}[p])$ ▷ only **one** action
37:             Adapt NN by backpropagating error $\delta = \boldsymbol{r}[p] - Q(\boldsymbol{s}_{last}[p], \boldsymbol{a}_{last}[p])$

---

where no moves are made but all players p perform an update towards their final reward $r[p]$. This speeds up learning, because the positive reward from a player making a winning move is directly propagated to its previous state.[8]

In Sec. 2.6 we stated that SARSA and Q-learning cannot benefit from the usage of afterstates. Why is it not possible to use in $Q(s, a)$ afterstates s' instead of states s? – When these algorithm have to decide which action to take, they cannot rely on the afterstate s', but they have to use the next state s''. This is because the random element added by the environment may have important influence on the action to take.

A.3   Eligibility Method

The general TD(λ)UPDATE rule for weights w_t is [16]:

$$
\begin{aligned}
w_{t+1} &= w_t + \alpha\delta_t \sum_{k=t_0}^{t} \lambda^{t-k} \nabla_{w_k} V_{w_k}(s_k) \\
&= w_t + \alpha\delta_t \nabla_{w_t} V_{w_t}(s_t) \\
&\quad + \alpha\delta_t \nabla_{w_{t-1}} V_{w_{t-1}}(s_{t-1})\lambda \\
&\quad + \alpha\delta_t \nabla_{w_{t-2}} V_{w_{t-2}}(s_{t-2})\lambda^2 \\
&\quad + \cdots
\end{aligned}
\tag{1}
$$

where in the last equation the sum is written out from index t *downwards* to index t_0.

Normally, this formula is realized in TD(λ) algorithms with the help of eligibility traces ([8,16,18]) and $t_0 = 0$ is used. But eligibility traces are unpractical (prohibitively slow) for games like 2048 (long episodes, millions of weights would need repeated updates[9]).

Jaskowski [6] has the brilliant idea to use a finite horizon $t_0 = \max(t - h, 0)$, that is, we use from the TD(λ)-equation (1) only the first $h + 1$ terms.[10] If we choose $h = \lfloor log_\lambda(0.1) \rfloor$ then we retain only those terms with $\lambda^{t-k} \geq 0.1$. This realizes a constant and relatively small computational effort, irrespective of how

---

[8] The second part of FINALADAPTAGENTS in Algorithm 2, where the final V is adapted towards 0, is not needed in Algorithm 5, because the Q-value of a terminal state is never used in SARSA.

[9] As Jaskowski [6] writes: „The eligibility traces vector is of the same length as the weight vector. The standard Sutton's implementation consists in updating all the elements of the vector at each learning step. This is computationally infeasible when dealing with hundreds of millions of weights." Thill [18] reduced this to the number of weights *activated* during an episode. Although this is sensible for ConnectFour (at most 42 moves; it was shown that each episode would not activate more than approximately 5000 weights), it is not viable for 2048: Since in 2048 an episode can last for several 10.000 moves, this may activate nearly all eligibility traces in the network. This would make an update step prohibitively slow.

[10] In the beginning, if $t < h$, it might be even less terms.

long an episode is. Note that episodes in the game 2048 can often have more than 10.000 moves.[11]

For n-tuple systems the value function in afterstate s' (for player $p(s')$) is

$$V(s') = \sigma\left(\sum_{i=1}^{m} \sum_{q \in S(s')} w_i[Ind_i(q)]\right) =: \sigma\left(\nu(s')\right) \tag{2}$$

with $S(s')$: the set of all states symmetric to s' (including s' itself), $N_S = |S(s')|$, $Ind_i(q)$: the index into the look-up table w_i of n-tuple i given state q, and m being the number of n-tuples. The term $\nu(s')$ (the double sum) is the activation of the n-tuple network induced by state s'.

If we take in Eq. (2) the derivative of $V(s')$ w.r.t. weight w_j, it is either 0 (if w_j is not activated by the current state s') or it is

$$\frac{\partial}{\partial w_j} V(s') = \sigma'\left(\nu(s')\right) \cdot 1$$

if w_j is activated by the current state s'. Thus the update function Eq. (1) reduces in the special case $\lambda = 0$ and $V(s')$ according to Eq. (2) to

$$w_i[Ind_i(q)] = w_i[Ind_i(q)] + \frac{1}{mN_S} \alpha \delta_t \sigma'\left(\nu(s')\right) \quad \text{with } q \in S(s') \tag{3}$$

with error signal $\delta_t = T - V(s')$, target T and $\sigma'\left(\nu(s')\right)$ being the derivative of sigmoid $\sigma()$ w.r.t. the activation induced by afterstate s'.[12] The new scaling factor $1/(mN_S)$, introduced by [6], ensures that for a linear net ($\sigma = id$) an update with $\alpha = 1$ moves $V(s')$ directly to the target value T, irrespective of the number of n-tuples and number of equivalent states.[13]

During an update of equivalent states, each n-tuple index is updated not more than once. See Sec. A.4 for more details.

The algorithm TD(λ)UPDATE implementing this is shown below in Algorithm 7.

---

[11] To build in the game 2048 a „16384“ tile, we need at least $16384/2 = 8092$ tiles „2“. In 90% of the moves we get a new tile „2“ (only 10% of the moves have a new tile „4“). Thus, 8192 moves is a rough estimate for an episode ending with a „16384“-tile.

[12] This means $\sigma'\left(\nu(s')\right) = 1 - V^2(s')$ in the case of $\sigma = \tanh$ and $\sigma'\left(\nu(s')\right) = 1$ in the case of the identity function $\sigma = id$.

[13] This can be shown by inserting Eq. (3) into Eq. (2):

$$\begin{aligned} V^{(new)}(s') &= \sum_{i=1}^{m} \sum_{q \in S(s')} \left(w_i[Ind_i(q)] + \frac{\alpha}{mN_S}\left(T - V(s')\right)\right) \\ &= \sum_{i=1}^{m} \sum_{q \in S(s')} \left(w_i[Ind_i(q)]\right) + mN_S \frac{\alpha}{mN_S}\left(T - V(s')\right) \\ &= V(s') + \alpha\left(T - V(s')\right) \\ &= \alpha T + (1-\alpha)V(s') \quad \xrightarrow[\alpha \to 1]{} \quad T \end{aligned} \tag{4}$$

Algorithm 6 MAKEACTION: Take action a in state s and observe cumulative reward tuple R, afterstate s' and next state s''.

1: **function** MAKEACTION(s, a)
2:     $s' \leftarrow$ COMPUTEAFTERSTATE(s, a)
3:     $(s'', R) \leftarrow$ ADDRANDOMPART(s') ▷ $R = (R(s''|p) \mid p = 0, \dots, N-1)$
4:     **if** $!T_{after}$ **then**
5:         $s' \leftarrow s''$ ▷ afterstate and next state shall be the same
6:     **return** (R, s', s'')

Algorithm 7 TD(λ)UPDATE. Inputs: δ_t is the delta signal for the actual afterstate s'_t. Additional inputs are: T_{lrnRM} (see Table 2), step size α, eligibility rate λ, number m of n-tuples, horizon h, and the h afterstates preceding s'_t. The variables $S(s'_k)$, N_S, $\sigma'(\nu(s'_k))$ and $Ind_i(s')$ are explained in Sec. A.3 (text around Eqs. (2) and (3)).

1: **function** TD(λ)UPDATE(s'_t, δ_t)
2:     **if** $(!T_{lrnRM}$ & s'_t generated by random action) **then return**
3:         **for** $k = t$ downto $\max(t - h, 0)$ **do**   ▷ for all states in s'_t's horizon
4:             **for** $i = 1$ to m **do**   ▷ for all n-tuples
5:                 $M \leftarrow \{\}$
6:                 **for** $s' \in S(s'_k)$ **do**
7:                     **if** $Ind_i(s') \notin M$ **then**
8:                         $w_i[Ind_i(s')] \leftarrow w_i[Ind_i(s')] + \frac{1}{mN_S}\alpha\delta_t\lambda^{t-k}\sigma'(\nu(s'_k))$
9:                         $M \leftarrow M \cup \{Ind_i(s')\}$ ▷ inhibit index $Ind_i(s')$

A.4   Helper Algorithms 6 (MakeAction) and 7 (TD(λ)Update)

Some remarks on the helper algorithms MAKEACTION and TD(λ)UPDATE:

– MAKEACTION is called once per loop in Algorithms 2 and 5 above.
– TD(λ)UPDATE$(s'_t, \delta, 0)$ is called each time we write *'Adapt NN by backprop-agating error δ ...'* in the Algorithms 2 and 5 above.
– The reason for the index set M in lines 5, 7 and 9 of Algorithm 7 is as follows: Each state $s' \in S(s'_k)$ activates exactly one weight $w_i[Ind_i(s')]$" in each n-tuple. But multiple equivalent states s' may activate the same weight w_i repeatedly. This may lead to a too large update step in line 8 of Algorithm 7. To avoid this, we keep in M a set of already visited indices $Ind_i(s')$ and skip the update step, if the weight was updated before.

The relevant classes implementing the new TD and SARSA algorithms are `TDNTuple3Agt` and `SarsaAgt`, resp.[14]

Some elements of the algorithms and their counter part in the GBG source codeare listed in Table 3.

---

[14] The older class `TDNTuple2Agt` [1,8] is now deprecated.

Table 3. Symbols of the algorithms and their source code equivalent in GBG

symbol	source code equivalent
MAKEACTION	constructor `NextState.NextState(state,action)`
COMPUTEAFTERSTATE	`StateObservation.advanceDeterministic()`
ADDRANDOMPART	`StateObservation.advanceNondeterministic()`
TD(λ)UPDATE	`NTuple2ValueFunc.update()`
TD-FARL-EPISODE	`TDNTuple3Agt.trainAgent()`
TD-algo from [1,8]	`TDNTuple2Agt.trainAgent()` (deprecated)
SARSA-FARL-EPISODE	`SarsaAgt.trainAgent()`
T_{after}	`AFTERSTATE, TDNTuple3Agt.getAFTERSTATE()`
T_{lrnRM}	`learnFromRM` in `TDNTuple3Agt.trainAgent()`

A.5   Temporal Coherence Learning (TCL)

The TCL algorithm developed by Beal and Smith [2] is an extension of TD learning. It has an adjustable learning rate α_i for every weight w_i and a global constant α. TCL works by replacing the global learning rate α in Step 8 of Algorithm 7 for each weight by the weight-individual product $\alpha\alpha_i$. The main idea behind TCL is pretty simple: For each weight, two counters N_i and A_i accumulate the sum of weight changes and sum of absolute weight changes. If all weight changes have the same sign, then $\alpha_i = |N_i|/A_i = 1$ and the learning rate stays at its upper bound. If weight changes have alternating signs, then $\alpha_i = |N_i|/A_i \to 0$ for $t \to \infty$, and the learning rate will be largely reduced for this weight.

The full TCL algorithm with all its variants is found in [1]. Of particular importance is the variant TCL-EXP that we newly developed in [1]: We introduced an exponential transfer function

$$g(x) = e^{\beta(x-1)} \tag{5}$$

and set $\alpha_i = g(|N_i|/A_i)$. It was shown that TCL-EXP results on the game ConnectFour in much faster learning and higher win rates than plain TCL [1].

A.6   Parameter Settings

In this appendix we list all parameter settings for the experiments in Tab. 1. Parameters were manually tuned with two goals in mind: (a) to reach high-quality results and (b) to reach stable (robust) performance when conducting multiple training runs with different random seeds. The detailed meaning of all parameters is explained in the GBG documentation [10].

2048: $\epsilon = 0$, $\alpha = 1.0$, $\lambda = 0$, NO output sigmoid, USESYMMETRY, AFTERSTATE, NORMALIZE=false, FIXEDNTUPLEMODE=2 [6, Fig. 3c, 4 6-tuple], TC-id with TC-Init= 10^{-4} and recommended weight-change accumulation. 200.000 training episodes, numEval=5.000.

TicTacToe: $\epsilon = 0.1 \rightarrow 0.0$, $\alpha = 1.0 \rightarrow 0.5$, $\lambda = 0$, output sigmoid, USE-SYMMETRY, NORMALIZE=false, one random 9-tuple, no TCL. 30.000 training episodes, numEval=1.000.

ConnectFour: $\epsilon = 0.1 \rightarrow 0.0$, $\alpha = 3.7$, $\lambda = 0$, output sigmoid, USESYMMETRY, NORMALIZE=false, fixed-n-tuple mode 1: 70 8-tuples. TC-EXP with TC-INIT= 10^{-4}, $\beta = 2.7$ and recommended weight-change accumulation. 5.000.000 training episodes, numEval=100.000.

Hex 6x6: In our experiments, Hex is played on a 6x6 hexagonal board.
$\epsilon = 0.2$, $\alpha = 1.0 \rightarrow 0.5$, $\lambda = 0$, output sigmoid, USESYMMETRY, NORMALIZE=false, 25 random 6-tuples, TC-id with TC-Init= 10^{-4} and recommended weight-change accumulation. Choose-Start-01 and Learn-from-RM active. 300.000 training episodes, numEval=10.000.

Othello 8x8: $\epsilon = 0.2$, $\alpha = 0.2$, $\lambda = 0.5$, horizon cut 0.01, eligibility trace ET, output sigmoid, USESYMMETRY, fixed n-tuple mode 6: 50 6-tuples. TC-EXP with TC-INIT= 10^{-4}, $\beta = 2.7$ and recommended weight-change accumulation. Choose-Start-01 active. 250.000 training episodes, numEval=2.000.
Evaluation is done by playing episodes from all 244 4-ply start states in both roles against selected opponents. If both agents played perfect, the expected win rate would be 50%.

Nim 3x5: $\epsilon = 0.1$, $\alpha = 0.5$, $\lambda = 0.5$, horizon cut 0.1, eligibility trace ET, output sigmoid, n-tuple fixed mode 1: one 3-tuple. TC-id with TC-Init= 10^{-4} and recommended weight-change accumulation. Choose-Start-01 inactive. Learn-from-RM active. 20.000 training episodes, numEval=1.000.

Nim3P 3x5: In our experiments, Nim3P (3-player variant) is played with 3 heaps of size 5 (3x5) and the extra rule being active.
$\epsilon = 0.15$, $\alpha = 0.2$, $\lambda = 0.5$, horizon cut 0.01, eligibility trace with RESET, output sigmoid, n-tuple fixed mode 2: two 3-tuples. TC-id with TC-Init= 10^{-4} and recommended weight-change accumulation. Choose-Start-01 active. Learn-from-RM inactive. 300.000 training episodes, numEval=10.000.