Reinforcement Learning for N-player Games: The Importance of Final Adaptation

Wolfgang Konen(✉) and Samineh Bagheri

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

Abstract. This paper covers n-tuple-based reinforcement learning (RL) algorithms for games. We present a new algorithm for temporal difference (TD) learning which works 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 this algorithm. 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 · 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.

© Springer Nature Switzerland AG 2020
B. Filipič et al. (Eds.): BIOMA 2020, LNCS 12438, pp. 84–96, 2020.
https://doi.org/10.1007/978-3-030-63710-1_7

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 n-tuple-based RL algorithm. N-tuples were introduced by Lucas [13] to the field of game learning. Our new learning algorithms extend the work described in [1,8,19] 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 [1,19], Othello [13], EinStein würfelt nicht [3], 2048 [18], 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 [14]. 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 AlphaBeta). 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.

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 (Sect. 2.5) uses the value function $V(s_t)$ which is the expected sum of future rewards when being in state s_t.

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

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

The nomenclature and algorithmic description follows as closely as possible the descriptions given in [6,18]. 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, in this paper we present a unified TD-update scheme inspired by [15] which works for 1-, 2-, ..., N-player games.

Our new RL-algorithm is partly inspired by [6,15] 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 Sect. 2.2.
- Eligibility method with horizon [6], see Sect. 2.3.
- Generalization to N-player games with arbitrary N [15], see Sect. 2.4.
- Final adaptation RL (FARL) for all players, see Sect. 2.6.
- Weight-individual learning rates via temporal coherence learning (TCL) [1,2].

More details are described in an extended technical report [11].

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 [13], although n-tuples were introduced already in 1959 for character recognition purposes. The remarkable success of n-tuples in learning to play Othello [13] 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 Support Vector Machines (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, Jaśkowski et al. [6,18] describe a clever mechanism to reduce the complexity of the value function $V(s)$.

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 s', but $2n$ possible

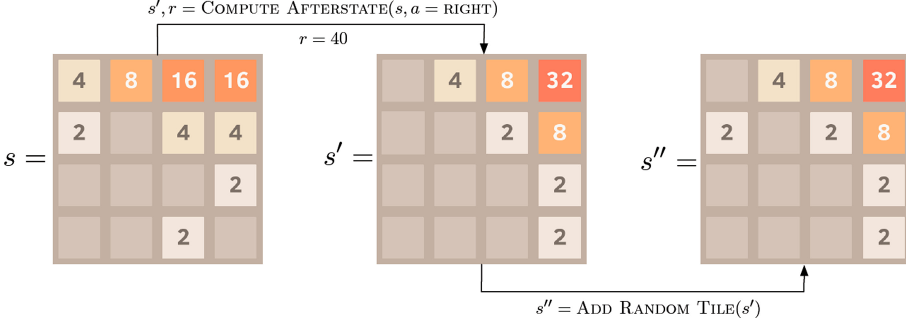

Fig. 1. For nondeterministic games it is better to split a state transition from s to s'' in a deterministic part, resulting in **afterstate** s', and a random part resulting in **next state** s'' (taken from [18]).

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''.

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 mapping from afterstates to game values (as we do in TD-learning, Sect. 2.5).

2.3 Eligibility Method

Instead of Sutton's eligibility traces [17] we use in this paper Jaśkowski'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 of the extended technical report [11] 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.

[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.

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 ← 0
 3:     repeat
 4:         Choose for player p_t action a_t ∈ A_t from s_t using policy derived from V
 5:                 ▷ e.g. ϵ-greedy: with probability ϵ random, with prob. 1 − ϵ 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 + γV(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 δ = V^new(s'_{t−1}) − V(s_{t−1})
10:         t ← t + 1
11:         s_t ← s''
12:     until s'' is terminal
```

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-player game (maximize next state's value) or a 2-player game (minimize next state's value). 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, van der Ree and Wiering [15] 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.

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 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

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 (number of players).

```
1: function TD-FARL-EPISODE(s_0)
2:     t ← 0;
3:     s_last[p] ← null   ∀ player p = 0, ..., 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:         (r, s', s'') ← MAKEACTION(s_t, a_t)      ▷ s': afterstate (after taking a_t)
8:                 ▷ r is the delta reward tuple from the perspective of all players p
9:         ADAPTAGENTV(s_last[p_t], r[p_t], s')
10:        s_last[p_t] ← s'     ▷ the afterstate generated by p_t when taking action a_t
11:        t ← t + 1
12:        s_t ← s''
13:    until (s'' is terminal)
14:    FINALADAPTAGENTS(p_t, 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(s_last[p_t], r', s')
18:    if (s_last[p_t] ≠ null) then          ▷ Adapt V(s_last[p_t]) towards target T
19:        Target T = r' + γV(s') for afterstate s_last[p_t]
20:        Use NN to get V(s_last[p_t])
21:        Adapt NN by backpropagating error δ = T − V(s_last[p_t])
22:
23:                         ▷ Terminal update of value function for all players
24: function FINALADAPTAGENTS(p_t, r, s')
25:    for (p = 0, ..., N − 1, but p ≠ p_t) do
26:        if (s_last[p] ≠ null) then         ▷ Adapt V(s_last[p]) towards target r[p]
27:            Use NN to get V(s_last[p])
28:            Adapt NN by backpropagating error δ = r[p] − V(s_last[p])
29:                       ▷ Adapt V(s') → 0  (s': terminal afterstate of player p_t)
30:    Use NN to get V(s')
31:    Adapt NN by backpropagating error δ = 0 − V(s')
```

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 Sect. 2.6.

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 in [1,8].
2. The same algorithm is viable for **arbitrary** number of players.
3. There is no (or less) unwanted crosstalk because of too frequent updates (as it was the case for some variants in [1,8]).[3]
4. Since states are connected with states one round (and not one move) earlier, positive or negative rewards propagate back faster.

2.6 Final Adaptation RL (FARL)

Once an episode terminates, we have a delta reward tuple for all players. A drawback of the plain TD-algorithm 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) does: Collect for each player *his/her* 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.[4]

One might ask whether it is not a contradiction to Sect. 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 usually 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. Jaśkowski [6] proposes to make an adaptation step towards target 0 for all terminal states.

[3] 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.

[4] The target has 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).

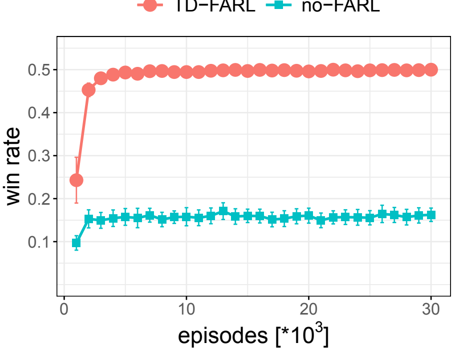

Fig. 2. Different versions of TD-learning on TicTacToe. Each agent is evaluated by playing games from different start positions in both roles, 1st and 2nd player, against the perfect-playing Max-N agent [12]. The best achievable result is 0.5, because Max-N will win at least in one of the both roles. Shown is the mean over 25 training runs. The error bars depict σ_{mean}.

3 Results

We show detailed 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 [1,11]. Note that due to TCL the effective learning rate adopted by most weights can be far smaller than α. The detailed parameter settings for all other games are given in the extended technical report [11].

3.1 TicTacToe

Figure 2 shows the learning curves of TD-learning. The red curve shows the full Algorithm 2 (TD-FARL-Episode). The blue curve shows the results when we switch off FinalAdaptAgents: The decrease in performance is drastic.

3.2 ConnectFour

Figure 3 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. [19] 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.

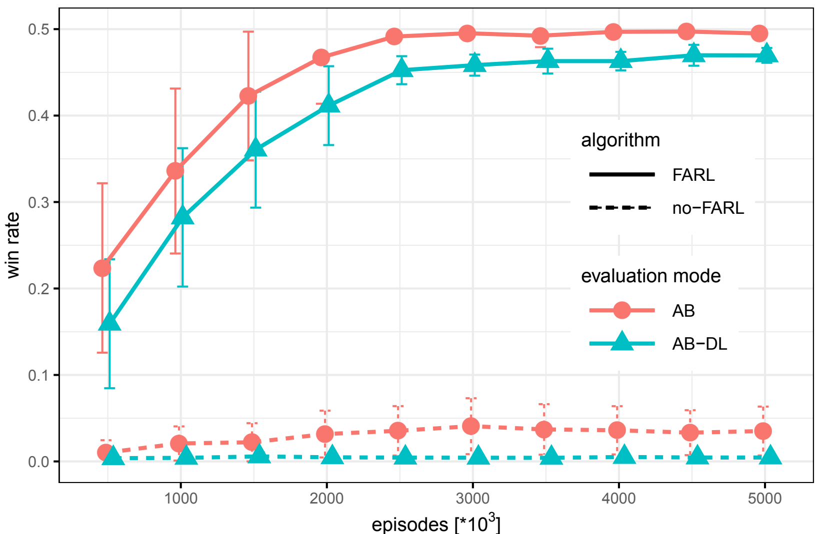

Fig. 3. TD-learning on ConnectFour. During training, agent TD-FARL is evaluated against the perfect-playing agents AlphaBeta (AB) and AlphaBeta-with-distant-losses (AB-DL). Both agents play in both roles (first or second). Since ConnectFour is a theoretical win for the starting player, the ideal win 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.

There is also a striking failure visible in Fig. 3: 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.[5] 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.

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} [12]	$49\% \pm 5\%$	$18\% \pm 6\%$		
ConnectFour	2	AB [19]	$49.5\% \pm 0.5\%$	$3.5\% \pm 0.1\%$	[4]	$0.0\% \pm 0.0\%$
		AB-DL [19]	$46.5\% \pm 0.5\%$	$0.0\% \pm 0.1\%$		
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 [15]	$95\% \pm 0.3\%$	$96\% \pm 0.2\%$	[15]	$87.1\% \pm 0.9\%$
Nim 3x5	2	Max-N_{15} [12]	$50\% \pm 1\%$	$12\% \pm 6\%$		
Nim3P 3x5	3	Max-N_{15} [12]	0.33 ± 0.03	0.03 ± 0.01		
		$MCTS_{5000}$	0.78 ± 0.02	0.09 ± 0.02		

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.

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,

[5] 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).

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 [12], being a perfect player for the games TicTacToe, Nim, Nim3P. For ConnectFour, AB and AB-DL [19] are perfect-playing agents introduced in Sect. 3.2. Edax_{d1} [5] is a strong Othello program, played here with depth 1. Bench [15] 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 [15] 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 [16] 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 [15]: 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 $\boldsymbol{s}_{last}[p_2]$ 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 Trans. Comput. Intell. AI 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.) International Joint Conference 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, December 2017
4. Dawson, R.: Learning to play Connect-4 with deep reinforcement learning (2020). https://codebox.net/pages/connect4. Accessed 21 Aug 2020
5. Delorme, R.: Edax, version 4.4 (2019). https://github.com/abulmo/edax-reversi. Accessed 1 Aug 2020
6. Jaśkowski, W.: Mastering 2048 with delayed temporal coherence learning, multistage weight promotion, redundant encoding, and carousel shaping. IEEE Trans. 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: Conference on Genetic and Evolutionary Computation, pp. 567–573 (2015)

8. Konen, W.: Reinforcement learning for board games: the temporal difference algorithm. Technical report, 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. Technical report, TH Köln (2020). http://www.gm.fh-koeln.de/ciopwebpub/Konen20a.d/TR-GBG.pdf
11. Konen, W., Bagheri, S.: Final adaptation reinforcement learning for N-player games. Technical report, TH Köln - Cologne University of Applied Sciences (2020). http://www.gm.fh-koeln.de/ciopwebpub/Konen20_TR.d/TR-FARL.pdf
12. Korf, R.E.: Multi-player alpha-beta pruning. Artif. Intell. **48**(1), 99–111 (1991)
13. Lucas, S.M.: Learning to play Othello with n-tuple systems. Aust. J. Intell. Inf. Process. **4**, 1–20 (2008)
14. 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
15. 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)
16. Silver, D., et al.: Mastering the game of Go without human knowledge. Nature **550**(7676), 354–359 (2017)
17. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998)
18. Szubert, M., Jaśkowski, W.: Temporal difference learning of n-tuple networks for the game 2048. In: 2014 IEEE Conference on Computational Intelligence and Games (CIG), pp. 1–8. IEEE (2014)
19. 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 Conference on Computational Intelligence in Games (CIG), Dortmund (2014)